2010:041

MASTER'S THESIS

802.11s based Wireless Mesh Network (WMN) test-bed

Luis Javier Sánchez Cuenca

Luleå University of Technology

Master Thesis, Continuation Courses
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of Computer Communication

2010:041 - ISSN: 1653-0187 - ISRN: LTU-PB-EX--10/041--SE

802.11s based Wireless Mesh Network (WMN) test-bed

Luis Javier Sánchez Cuenca

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
Communication Networks Research Group

March 2010

ABSTRACT

Wireless Mesh Networks (WMNs) are one of the key technologies that is likely to play an important role in wireless networking in the next decade. They will help to realize the long-lasting dream of network connectivity anywhere and anytime with simplicity and low cost. Their capability of self-organization significantly reduces the complexity of network deployment and maintenance, and thus, requires minimal upfront investment.

The main objective of this master thesis is to create a Wireless Mesh Network test-bed based on the project Open80211s in order to analyze the performance of this type of networks.

The goals of this master thesis are to build and verify two installation packages for evaluating 802.11s WMNs. The first package will be based on Ubuntu Linux using the open80211s implementation and the second one will be based on the *NS-3* network simulator. The verification includes performing tests on the performance and the quality of the network for typical topologies and traffic loads, and comparing the results between both systems and with the corresponding theoretic values.

The main deliveries and results are two systems where we can test WMN technology. On one side, there will be a WMN test-bed in which real conditions can be analyzed, and on the other side there will be a WMN in which it will be possible to simulate a topology and check whether the results agree with reality.

Preface

This work has been carried out between September 2009 and March 2010 at Luleå University of Technology in Luleå (Sweden).

I would like to dedicate this work to my grandfather, José Sánchez Pérez, he showed me a way of life and he always will be my reference. Wherever you are, thank you very much, I love you, “yayo”.

I would also like to thank the people at Communication Networks Research Group for helping me with this master thesis. Particularly, I thank my supervisor, Ulf Bodin, for helping me all the way, for bearing with all my questions and for sharing his knowledge.

A special thank you goes to all my friends, from Luleå and from Spain, for cheering me up during the bad moments.

And last, but not least, an enormous thank to my family for giving me the chance to study and develop my Master Thesis in Luleå so far from them. There are no words to express my love and gratitude.

Luleå – March 2010
Luis Javier Sánchez Cuenca

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This document is the Thesis report titled “802.11s based Wireless Mesh Network (WMN) test-bed”. This project has been developed by Luis Javier Sánchez Cuenca and supervised by Ulf Bodin from the Communication Networks research group of Luleå University of Technology.

1.1 Background

Nowadays, there is high interest in the application of the multi-hop wireless communications known as Wireless Mesh Networks (WMNs).

Wireless mesh networks consist of mesh routers and mesh clients, where mesh routers have minimal mobility and form the backbone of WMNs. They provide network access for both mesh and conventional clients. The integration of WMNs with other networks such as the Internet, cellular, IEEE 802.16, IEEE 802.11, IEEE 802.15 [1], sensor networks, etc., can be accomplished through the gateway and bridging functions in the mesh routers. Mesh clients can be either stationary or mobile, and can form a client mesh network among themselves and with mesh routers.

The high variety of manufacturers that are producing products for WMN indicates the increasing interests of the industry in this topic. Furthermore, the main groups of standardization are defining WMN standards [2, 3] which will allow a better interoperability between these networks.

The IEEE 802.16 working group develops the physical layer and medium access control sublayer standards. Inside this working group, there are several task groups:

- IEEE 802.16a: In charge of adding a “mesh mode” to the Point-to-Multi-Point (PMP) architecture.

- IEEE 802.16b: Providing a QoS (Quality of Service) feature.
- IEEE 802.16c: Supporting interoperability with protocols and test-suite structures.
- IEEE 802.16d: Providing extensions to physical layer for developing access.
- IEEE 802.16e: To enhance the mobility of Mobile Stations (MSs).
- IEEE 802.16f: Supporting multi-hop functionality.
- IEEE 802.16g: Providing efficient handover and QoS.

The IEEE 802.11s tasking group develops the specification of a new protocol suite for the installation, configuration, and operation of a WLAN mesh. Its implementation works with the existing physical layer of IEEE 802.11a/b/g/n and includes the extensions in topology formation to make the WLAN mesh self-configuration possible.

The IEEE 802.15 specifies the physical layer and medium access control sublayer functions of Wireless Personal Area Network (WPAN). Specifically, IEEE 802.15.5 provides an architectural framework for interoperable, stable, and scalable wireless mesh topologies for WPAN devices.

Some of the principal reasons that motive the development of this technology are:

- Higher capacity at less cost because it is demonstrated that the capacity of a wireless network can be improved by using repeaters (knowing the distance and interferences between nodes) [4] .
- Easy deployment because one of the most important features of this type of networks is that they are self-configuring. This characteristic makes this type of network ideal to be used in emergencies or in natural catastrophes.
- High independence (for example self-configuring, self-repairing of routes). This fact helps the maintenance of the net.

The above-mentioned reasons makes WMN a very good solution to provide Internet access (or any other Network access) at public locations as airports, small villages, and places where it is difficult and very expensive to install a wired network topology.

Wireless access based on the IEEE 802.11 standard are becoming increasingly common, both at homes and for hot-spots at airports, Internet cafés and other public locations where wireless Internet access is desired. The IEEE is now in the process of finishing the work on 802.11s, which is an extension to 802.11 for wireless mesh networking (i.e. multi-hop wireless networks).

802.11s facilitates two-tier wireless infrastructures, where the lower-tier provides access for clients to the upper-tier of the wireless infrastructure. The upper-tier constitutes the wireless backhaul. Some nodes of this wireless backhaul are gateways to wired networks such as the Internet, enterprise networks, or to whatever infrastructure the WMN provides access.

The open80211s consortium [5] is developing a reference implementation of 802.11s, which is included in the native Linux kernel and in some Linux distributions such as Ubuntu [6]. Efforts are also made by IITP [7], a research institute in Russia, to include support for 802.11s in the network simulator version 3 (*NS-3*) [8].

1.2 Objectives of the Thesis

The first objective of this master's thesis is to build and verify an installation package for running 802.11s WMNs based on PCs within Ubuntu Operating System. The verification includes performing quality network tests for typical topologies and traffic loads, and comparing the results with the corresponding theoretic values. Metrics that shall be examined are to be decided as part of the project. The goal for this part is a test-bed for 802.11s that is verified and easy to setup for future experiments.

Related to this objective, we have the following goals:

1. **Install Ubuntu:** Choose a version of the Ubuntu Operative System and install it on the computers, verifying that everything is correct.
2. **Analyze Hardware:** Analyze and search which is the hardware that allows us to configure a Wireless Mesh Network on our Ubuntu version and install it on each computer.
3. **Configure a Wireless Mesh Network:** After installing all the Wi-Fi cards, configure a WMN according to the 802.11s standards [5]. Start configuring a pair of computers and, after that, add computers singly to the Network, up to eight computers. Test the Network according to the next point.
4. **Network Test:** Decide which are the tests used to verify that the Network work correctly.
5. **Obtain results:** Extract conclusions about all the test done.

The second objective is to establish in *NS-3* the same scenarios tested in reality, and to compare the results obtained in the simulator with those of the real 802.11s test-bed. This means to build and verify a similar installation package as done for this test-bed. In case any discrepancies and/or weaknesses are identified between the real test-bed and

the simulated WMN, these shall be properly documented. The goal for this part is a verified simulation environment that corresponds to the 802.11s test-bed.

Related to the second objective, we have the following goals:

1. **Configure NS-3:** Learn how to use *NS-3* and configure it to establish the same scenarios that we have tested in the real world (Network of the first part of this project).

2. **Obtain results:** Simulate the Network and compare the results obtained from simulations with the ones obtained with the test-bed, analyzing possible differences.

1.3 Limitations

The research presented in this thesis intends to create a test-bed based on the IEEE 802.11s protocol and study its behaviour. These are the limitations for this thesis:

- The WMN will have eight computers. This way, it is big enough to allow useful studies of its behaviour while being reasonably easy to configure. It should be possible to add more computers to the network without having to make many changes.

- The Test-bed will be configured manually, assigning static and local IP addresses to each computer of the network.

- The Test-bed shall be tested in a specific topology, describing environment conditions and the specifics parameters of the network.

- The *NS-3* simulations should adapt to the real conditions using classes and objects already implemented in *NS-3*. New protocols or models will not be developed.

1.4 Structure of the Thesis

This thesis starts with a summary on what we have done. This first Chapter shortly introduces the Wireless Mesh Network technology and enumerates the tools and equipment we have used. It also lays out the time plan of the Thesis.

Chapter 2 focuses on the first goal of this Thesis, the WMN test-bed, planning how we have done it and explaining the steps we have followed to make it possible. It also shows how we have done the installation and configuration of the test-bed, which topologies we have chosen, on which scenarios we have test them, and the results of that test are explained at the end of the chapter.

Next, in Chapter 3, the thesis gives an overview of *NS-3*, the second goal of this thesis. Firstly, the main *NS-3* concepts are introduced. Secondly, laid out the implementation decisions. Finally, experiments and results obtained from these experiments are explained.

Chapter 4 shows the comparison of the results obtained in the test-bed and in the *NS-3* simulation. Finally, chapter 5 displays the conclusions drawn in the thesis work and Chapter 6 sketches work that can be conducted to continue this thesis work.

1.5 Analysis of the results

When we examined the experiments that we did in the test-bed and in *NS-3* we focussed on analyzing the throughput.

Throughput is the average rate of successful message delivery over a communication channel and is usually measured in bits per second (bit/s or bps, in some experiments we used Mbps).

We used different formulas to calculate the throughput depending on whether we sent UDP or TCP traffic. For UDP we used:

$$Throughput = \frac{TotalDataReceived}{TotalTime}$$

Where:

- **Total Data Received:** Bits received (UDP packets) by the destination from the source.
- **Total Time:** Time elapsed since the source sends the first packet until the source sends the last packet.

And for TCP we used the next formula:

$$Throughput = \frac{TotalDataTransmitted + TotalDataReceived}{TotalTime}$$

Where:

- **Total Data Transmitted:** Bits transmitted (TCP packets) from the source to the destination.
- **Total Data Received:** Bits received (ACK) by the source from the destination.

- **Total Time:** Time elapsed since the source sends the first packet until the source receives the last ACK.

Both throughputs were comparable because we were measuring the total network UDP/TCP traffic. That is why when we calculate the UDP throughput we only the used total data transmitted (UDP sends packets without received any acknowledgment) and when we were calculating the TCP throughput we also take into account the acknowledgment data.

We also calculated the Confidence Interval for the throughput, according to the next formula:

$$Confidence = \frac{N \cdot \sigma}{\sqrt[2]{n}}$$

And the Confidence Interval is ($\overline{X}$ - Confidence, $\overline{X}$ + Confidence)
Where:

- **N:** Normal Distribution value according to the confident level chosen.
- σ**:** Standard deviation of samples of throughput.
- **n:** Number of samples of throughput.
- $\overline{X}$**:** Throughput samples average.

1.6 Project Time Plan

In this section it is shown the time plan of this project.

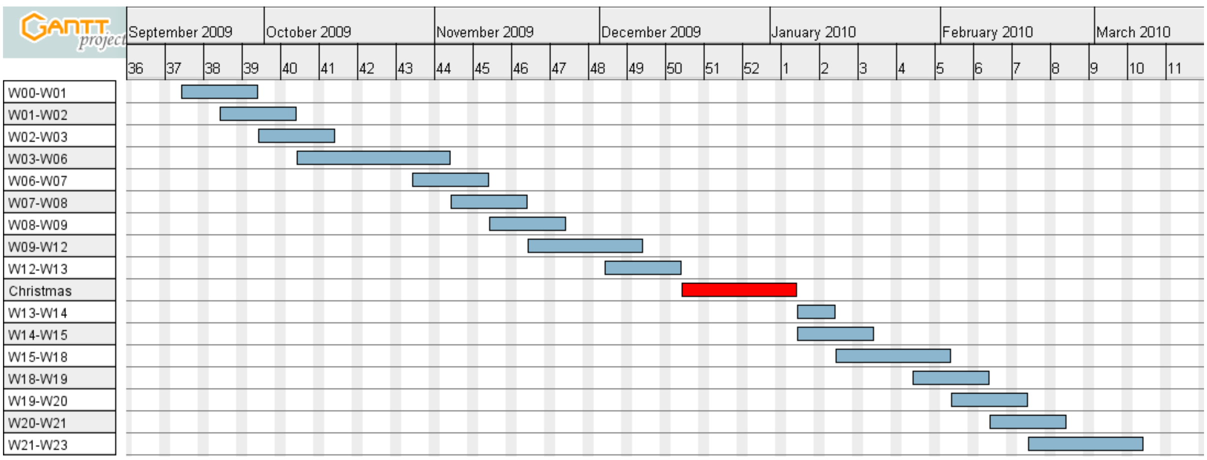

Figure 1.1: Project Gantt's Diagram

As we can see in the Project Gantt's Diagram (*Figure 1.1*), the project work is divided in weeks of full time work and with 40 hours per week. This work is explained in the following list.

- **W00-W01:** Study relevant the 802.11 standards, usage of Linux experimental packages, identify hardware requirements for the test-bed, and order missing parts.
- **W01-W02:** Document background, objectives and delimitations for the master thesis.
- **W02-W03:** Identify basic functionally and metrics that shall be tested to verify a simple 802.11s WMN.
- **W03-W06:** Build an installation package for setting up Ubuntu with 802.11s on two PCs, and test basic functionality and metrics for this installation.
- **W06-W07:** Demonstrate the basic installation and testing.
- **W07-W08:** Document the installation package and the basic testing for the master thesis.
- **W08-W09:** Identify additional functionality and metrics for extended tests of larger WMNs consisting of up to 8 nodes.
- **W09-W12:** Extend the installation package for arbitrary sizes of test-beds, and test additional functionality and metrics for this installation (8 PCs will be made available for this phase).
- **W12-W13:** Demonstrate the extended installation and testing.
- **W13-W14:** Document the extended installation package and the additional testing for the master thesis.
- **W14-W15:** Study *NS-3* and the 802.11s implementation in that environment.
- **W15-W18:** Build an installation package for testing the same scenarios as tested in reality also in *NS-3*, test the same functionality and metrics for this installation as tested on the real-test-bed, and identify discrepancies and/or weaknesses.
- **W18-W19:** Demonstrate the *NS-3* installation and testing.
- **W19-W20:** Document the installation package and the *NS-3* testing for the master thesis.
- **W20-W21:** Correct discrepancies and/or weaknesses in *NS-3* to improve the simulation environment.
- **W21-W23:** Document the corrections in *NS-3* testing for the master thesis, and finish the thesis to make it ready for presentation and publication.

1.7 Technologies Used

In this secction we shows the technologies used to do this project.

1.7.1 Operating System

Ubuntu is a Linux-based operating system. Linux is a generic term referring to Unix-like computer operating systems based on the Linux kernel. Their development is one example of free and open source software collaboration.

Typically all the underlying source code can be used, freely modified, and redistributed, both commercially and non-commercially, by anyone under the terms of the GNU GPL and other free software licenses.

Ubuntu is chosen as operative system because:

- It is free of charge, so you do not pay any licensing fees. It can be downloaded, used and shared with other people without paying anything.
- It is possible to have always the latest applications that the open source world has to offer.
- On Ubuntu, the open80211s package that we need to configure our Wireless Mesh Network is implemented.

We used Ubuntu 9.04 [6]. It includes the latest enhancements and is maintained until the beginning of 2010. It comes with the 2.6.28 Linux-kernel but the kernel needs to be updated up to 2.6.29.

1.7.2 Network

To analyze the Network we used *Wireshark* [9] and *Iperf* (or *Jperf*, a *Iperf-java* application). The reason for choosing Wireshark was that it is the network protocol analyzer most commonly used to sufficiently support our goals. *Iperf* was selected because it is a commonly used network testing tool that can create TCP and UDP data streams and measure the throughput of the network that is carrying them. *Iperf* is a modern tool for network performance measurement written in C++.

To identify the best transmission channel, *Network Stumbler* offers the possibility to know who is using each channel and the signal/noise of those users. This is useful because we are interested in testing our network on a sparsely used channel.

Others tools that we used:

- **Ping:** Is a computer network administration utility used to test whether a particular host is reachable across an Internet Protocol (IP) network and to measure the round-trip time for packets send from the local host to a destination computer, including the local host's own interfaces.

- **Tcpdump:** Is a common packet analyzer that runs with the command line. It allows the user to intercept and display TCP/IP and other packets being transmitted or received over a network to which the computer is attached.

- **Route:** Is a tool that manipulates the kernel's IP routing tables. Its primary use is to set up static routes to specific hosts or networks via an interface.

- **Iptables:** Provide a table-based system for defining firewall rules that can filter or transform packets. It can be also used to create static MAC address routing.

- **Iw:** Is a new tool, still under development, used to configure utilities for wireless devices.

- **Iwconfig:** Used to set the parameters of the network interface which are specific to the wireless operations.

- **Ipconfig:** Is a utility that communicates with the IP configuration agent to retrieve and set IP configuration parameters.

The current stable release of *Wireshark* is 1.2.2, of *Iperf* is 2.0.8, of *Network Stumbler* 0.4.0 (Build 554), and of *tcpdump* is 3.9.8.

Mathematical calculations and data plotting are carried out in *Matlab*. To elaborate this report we have used LaTeX.

1.7.3 Hardware

The 8 computers used in the first part of this master thesis have the following characteristics:

Processor: Intel® Pentium® 4 CPU 2.40 GHz
Cache size: 512 KB Hard Disk Capacity: 120 GB
Ram Memory: 256 MB
Ports USB: 6 ports USB 1.1 (data transfer rate of 12 Mbit/s)

The computer used on the second part of the project it is a laptop with the next characteristics:

Processor: Intel® Core 2 duo
CPU P8600 2.40 GHz (2 CPUs)
Cache size: 512 KB
Hard Disk Capacity: 300 GB
Ram Memory: 3 GB

About the wireless card used, it is analyzed which one works with our requirements, if it is supported by the operating system used, and if it is work with the open80211s project.

CHAPTER 2

WMN Ubuntu Test-Bed

In this section it is explained how the Ubuntu Linux Operative System has been installed on the computers. It is also explained how the open80211s installation package has been developed and how to set up the Wireless Mesh Network. Finally, all the network testing process and the results obtained are described.

2.1 Operating System Installation

Ubuntu is an operating system built by a worldwide team of expert developers. Ubuntu is free of charge and everyone can download it [6], use and share the Ubuntu with everybody for nothing. Everybody can contribute to Ubuntu project by writing new software, packaging additional software, or fixing bugs in existing software.

For the test-bed, Ubuntu 9.04 version has been chosen because it was the latest version when this project started and because it gives all the tools needed to install the Open80211s package.

To analyze which is the best Wireless USB Adapter, we had to look for one working with Ubuntu 9.04 and supporting IEEE 802.11s. We were interested to know which wireless card is compatible with the OS and if the driver works with the Open80211s project. For that, we have analyzed some drivers and some kernels.

The first driver we analyzed was "zd1211rw" [10]. When we started to read about it, it seemed to work with 2.6.26 kernel version and we found a Wireless USB Adapter that works with this driver, but after some additional readings [11], we concluded that this driver has problems in some systems because of mesh beaconing triggers, which appears to be a firmware bug.

The second driver was "ath9k" [12] but we discarded it because this driver does not beacon with interfaces in Mesh Point mode, until the user performs a scan.

We tried other drivers (as it is shown in [13]), and finally we decided to use "p54" [14] because it works with Ad-Hoc, AP, mesh, monitor and station mode. It works with the 2.6.29 Linux-kernel, but as we are using Ubuntu 9.04, we had to upgrade the Linux-kernel from 2.6.28 to 2.6.29.

Once we decided which driver was the best option, we looked for a Wireless USB Adapter and we chose the *"D-Link DWL-G122 Wireless G USB Adapter"* [15], see *Figure 2.1*.

Figure 2.1: D-Link DWL-G122 Wireless G USB Adapter

This card has the following characteristics:

Standards supported: USB 2.0, IEEE 802.11b and IEEE 802.11g
Wireless Signal Rates: 54Mbps, 48Mbps, 36Mbps, 24Mbps, 18Mbps, 12Mbps, 11Mbps, 9Mbps, 6Mbps, 5.5Mbps, 2Mbps and 1Mbps (with automatic fallback).
Frequency Range: 2.4GHz to 2.462GHz
Operating Voltage: 5 VDC +/- 5
Receiver Sensitivity: 54Mbps OFDM, 48Mbps OFDM, 36Mbps OFDM, 24Mbps OFDM, 18Mbps OFDM, 12Mbps OFDM, 11Mbps OFDM, 9Mbps OFDM, 6Mbps OFDM, 5.5Mbps CCK, 2Mbps QPSK, 1Mbps BPSK
Transmit Output Power:

- 802.11b: +16dBm at 11, 5.5, 2, and 1Mbps
- 802.11g: +10dBm at 54 and 48Mbps +12dBm at 36 and 24Mbps +14dBm at 18, 12, 9 and 6Mbps

Power Consumption:

- Transmission: 310 mA max
- Reception: 290 mA

Security: 64/128-bit WEP and WPA-Wi-Fi Protected Access
Media Access Control: CSMA/CA with ACK
Wireless Signal Range:

- Indoors: Up to 328 ft (100 meters)
- Outdoors: Up to 1312 ft (400 meters)

Antenna Type: Omni Directional
Temperature:

- Operating: 32°F to 104°F (0°C to 40°C)
- Storing: 4°F to 167°F (-20°C to 75°C)

These parameters are very important because we must know the admitted range to set them on the network, and which one we can change to different values to know the performance of the network. For example, it is possible to change the data rate between 1 Mbps and 54 Mbps or to change the power transmission/reception between 0 and 310 mA and 290 mA, respectively.

2.2 Open80211s package Installation

Open80211s is a consortium of companies who are sponsoring (and collaborating in) the creation of an open-source implementation of the emerging IEEE 802.11s wireless mesh standard. Open80211s is a reference implementation of the upcoming 802.11s standard [16] on Linux. Open80211s is based on the mac80211 wireless stack [17] and should be run with any of the wireless cards that mac80211 supports.

The goal is to create an installation package working on Ubuntu, consisting of shell scripts, which allows installing Open80211s and lets creating a WMN with this technology.

There were two ways to install this system. The first one (called compact-wireless) consisted in downloading a package [18], already compiled, and installing it with the latest stable release. This package works with kernels equal or up to the 2.6.26 version. The second way (called wireless-testing), was to install the latest advances on the Linux

wireless subsystem [19], choosing the driver, building and compiling the kernel. For this option, we should have had updated the kernel up to 2.6.29 version. To complete this process the instructions described in *Appendix A* were followed.

After analyzing the options, we chosed the second one to install this system in the test-bed. This is because we want the system to be installed in the computer with the latest features and also because we are sure that it supports the *D-Link DWL-G122* driver and all features of the Open80211s package.

Once the latest version was downloaded, we started installing it, following recommendations made by the developers [20]. The main steps of the installation were:

1. Find and install the libraries needed to build the kernel packages.
2. Install the libraries needed to use the menu to configure kernel options.
3. Configure the kernel, choose the driver options and the IEEE 802.11s settings.
4. Build the kernel and make the packages.
5. Install the packages generated by the last step.
6. Install the tool *IW* required to setup a WMN.

Some problems were found because some of the libraries that the packages need were not clear and also because when we were building the kernel (that takes almost two hours and a half), it gave us some errors that we had to fix. For this reason, we have created a script to install the package. With it, anyone who wants to install this package just needs to follow the steps and wait the time that takes building and installing the kernel.

All the files needed to install the same test bed system are in a CD attached to this report, in a folder called “wireless testing”. To install it, it is only necessary to copy that folder to the computer home folder and follow the steps written on the *README-INSTALL* file. In Appendix B is possible to see this file and also the script developed to install the Open80211s package.

2.3 WMN setup

The first step to setup the WMN test-bed was installing the Open80211s package on the eight computers. Once the system was installed we had to configure each computer in the network. To setup the network we have developed a script that setting the parameters of the network allows us to configure it in a very fast and very easy way. To know how to use it, we follow the steps described in the *README-CONFIGURE* file (*Appendix B*).

It is also important to know that there are some parameters that should be kept in mind when setting up a Network of these characteristics, such as the transmission channel, the frequency, the transmission power, the bit-rate, etc. To set these parameters we used tools such as *Iw*, *Iwconfig* and *Ipconfig*. Those are very useful because they allow running the network with the parameters selected by the user. Networks parameters are explained in the next section.

2.4 WMN topology

In this project we have created a chain topology (as shown in Figure *2.2*) in two different places with different parameters to analyze the Network, which results in two different topologies. Both topologies were established in the basement of the A-Building at the main campus of Luleå University of Technology.

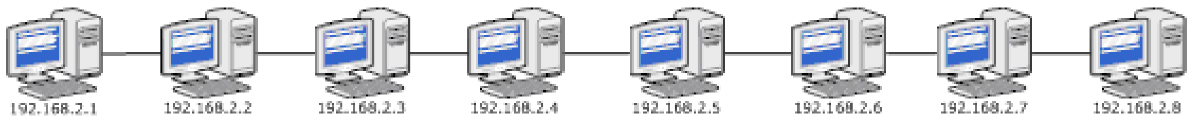

Figure 2.2: Chain WMN topology

Table 2.1 shows how an IP static address has been assigned to each computer in order to identify them. We are using as netmask 255.255.255.0 . From now on, we will refer a computer as a node of the network.

Computer ID	IP address
Proyecto 1	192.168.2.1
Proyecto 2	192.168.2.2
Proyecto 3	192.168.2.3
Proyecto 4	192.168.2.4
Proyecto 5	192.168.2.5
Proyecto 6	192.168.2.6
Proyecto 7	192.168.2.7
Proyecto 8	192.168.2.8

Table 2.1: Computers info (ID, IP)

2.5 WMN experiments

We made two experiments with the 8-computers test-bed, each of them with the same network topology (chain topology) but at different places and with different settings. In this section, the test-bed experiments conducted on this thesis are described.

2.5.1 Description of Experiment I

The topology created for this experiment consisted on the placement the nodes in four near rooms and one node in the corridor according to *Figure 2.3*. As it is shown, two nodes were in the first room, two other were in the nearest room, two other were in another room, and regarding the last two, one was in the corridor and the other one was in another room.

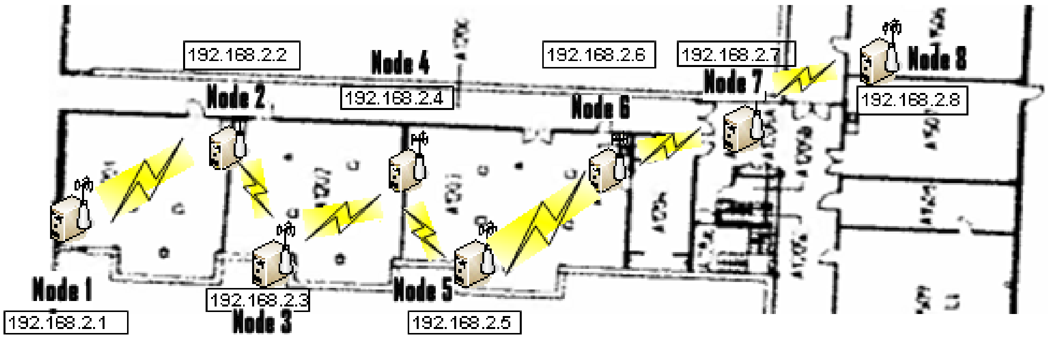

Figure 2.3: Topology of experiment I of the WMN test-bed

The distances between nodes are detailed in *Table 2.2*, but it is also important to note that there were walls between some of the nodes:

- Between node 2 and 3 one wall.
- Between node 4 and 5 one wall.
- Between node 6 and 7 two walls.
- Between node 7 and 8 one wall.

This information is important because the signal from antennas decreases when going through walls. In this topology, we want to have as low interference as possible between nodes, which is why we isolated the antennas of some nodes. To isolate antennas we have used each antenna with one empty can together with aluminium folio to wrap the

dongles so that only the cable comes out. To know if the RSS (Received Signal Strength) decreases when we use this isolation technique, we measure each RSS before and after applying this method. Using this technique we have managed to isolate only some nodes but it has been impossible to isolate all and obtain a network in which each node can only communicate directly with its neighbours.

	Distances
Node 1 → Node 2	9.20 m
Node 2 → Node 3	5.90 m
Node 3 → Node 4	7.70 m
Node 4 → Node 5	4.50 m
Node 5 → Node 6	11.43 m
Node 6 → Node 7	10.60 m
Node 7 → Node 8	8.00 m

Table 2.2: Distances between nodes on the topology of experiment I

Network parameters are defined in *Table 2.3*, where we can see the parameters set and the value used for the test.

Parameter	Value
Routing	Dynamic
Data rate	54 Mb/s
Transmitted signal power	18 dBm Channel
Frequency	2437 e6 (channel 6)
RTS/CTS	Off
Fragmentation threshold	Off
Power management	On

Table 2.3: Network parameters of the experiment I

Scenario

The scenario of this experiment (*Figure 2.4*) allows to know the performance of the test-bed, sending traffic from the first node, to the second, when finishing to the third, and so on up to the last one.

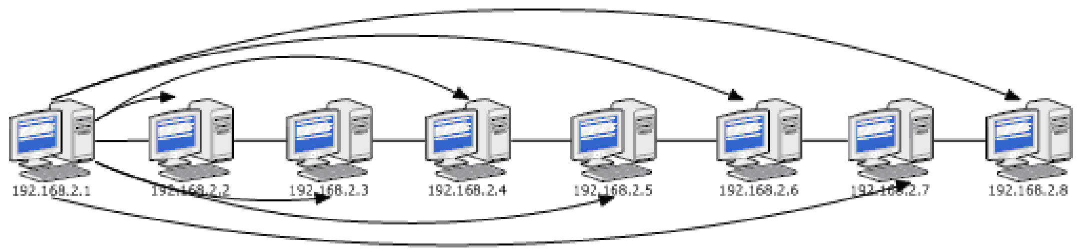

Figure 2.4: Network scenario experiment I

To do this experiment we sent traffic with the *Iperf traffic generator* using two transport protocols, UDP and TCP, setting the same scenario for both (details in *Table 2.4*).

Transport Protocol	UDP	Transport Protocol	TCP
Number of repetitions per node	5	Number of repetitions per node	5
Buffer size	41KB	Buffer length	2KB
Packet size	1.5KB	Maximum Segment size	1.5KB
Connection port	5001	Window size	56KB
		Connection port	5001

Table 2.4: Parameters of the experiment I

2.5.2 Results of experiment I

After applying the settings and parameters described in the previous sections, we have used *Jperf*, that is a graphical user interface using *Iperf*, that allows us to calculate the throughput of the network with the setting and parameters we have set for our test.

We started with the UDP test, and after finishing it, we did the TCP test with the same parameters described above. We can see the throughput results in Appendix D. We can see the throughput of the network when there is one hop, two hops, and so on until seven hops (the whole network, 8 nodes, 7 hops). With these results we plot two graphics (*Figure 2.5*) representing throughput values, and with them we can analyze the results of this experiment.

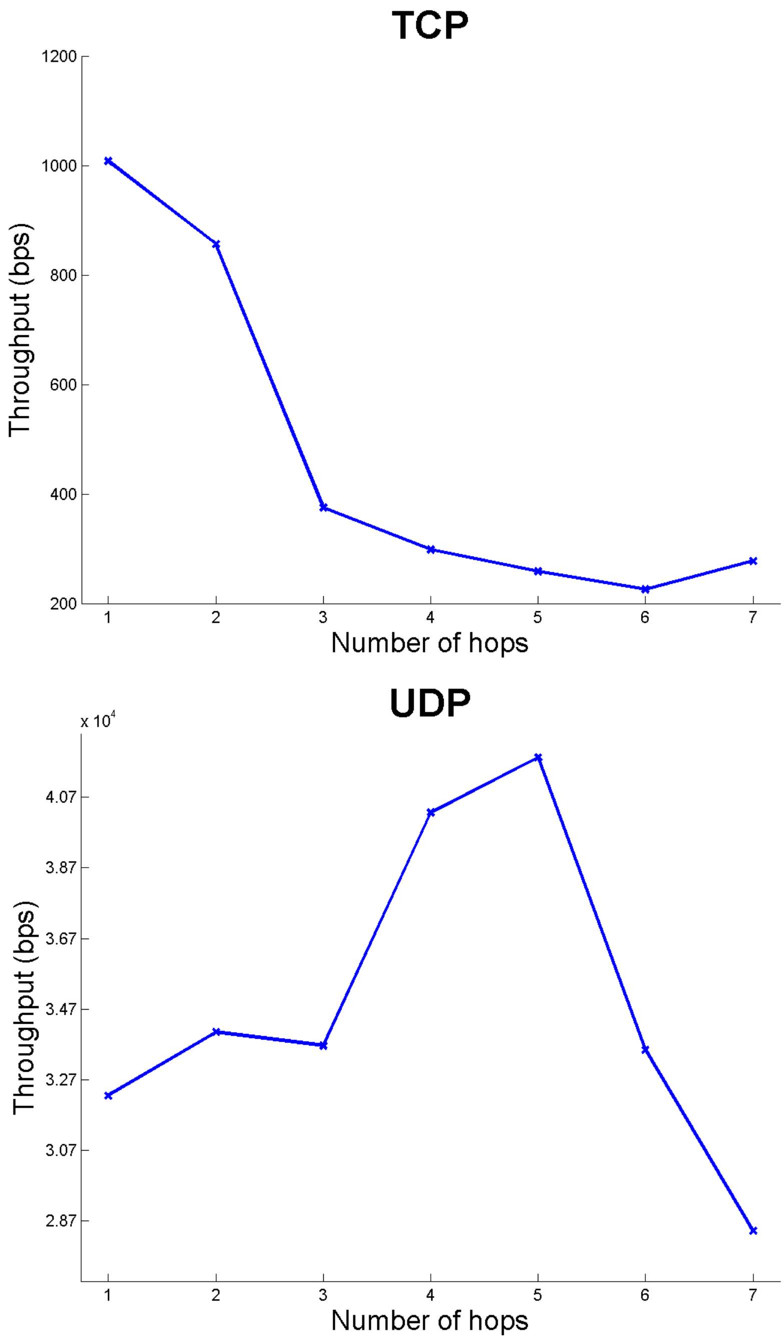

Figure 2.5: Throughput for UDP and TCP

From the UDP graphic we can see how the throughput increases from 3 hops to 6 hops. This behaviour indicates that something is wrong. The normal behaviour should be that if there are more hops (data should go accross more nodes) using the same channel

(frecuency) the throughput should decrease because interference between transmissions at different hops.In TCP we can also observe this behaviour but with no such big difference. From six to seven hops we can see how the throughput increases. The reason of this behaviour could be that nodes are not as much isolated as they should be or they can reach better other nodes, and hence they can connect to other nodes directly. Due to this fact, when they want to transmit, the network chooses the shortest way (it is working with dynamic routing) and does not make the expected hops. From these results we conclude that the network topology created is not a chain.

Another problem that we found it was that the data rate its 54 Mbit/s but it never achieved this throughput. Furthermore, in both tests, it always worked with a bandwidth below 12 Mbit/s. At first, we thought that fallback to lower transmission rate option was activated because the channel was too noisy. We used *Network Stumbler* to see if the channel we were using (channel 6) was as crowded of traffic as it looked. We detected that someone was using it, but not as much as having such big decrease of the rate, and we also checked that the *auto fallback* option was not active. After thinking about it, we found that our computers got USB 1.1 ports, and the antennas were connected to those ports. This means, according to the USB 1.1 spec, that a maximum data rate of 12 Megabits per second can be reached.

We can observe also that the throughput of the network it not very high. This is because there are too many walls between nodes and also because the aluminium folio and cans used on the antennas weaken the signal.

From this experiment we have obtained some important information of the network, and for the next experiments we changed some settings and parameters:

- To make sure that the topology of the network is a chain and nodes only communicates with their nearest nodes (maximum one on their right and one on their left) we used static routing.
- To avoid problems with USB ports, we set the data rate to 11 MBits/s. Also, before starting the experiment, we looked with *Network Stumbler* which one was the less crowded channel.
- To reduce the problem of the low throughput, we took out the aluminium folio and the cans from the antennas, and changed the place of the experiment, trying to make each node able to create a line of sight (with nothing between them) with the nearest nodes.

2.5.3 Description of Experiment II

The topology of this experiment consisted of putting the nodes of the network in corridors all around the building creating a chain topology but trying that walls did not act decreasing the signal between nodes. We can see in *Figure 2.6* the topology of this network.

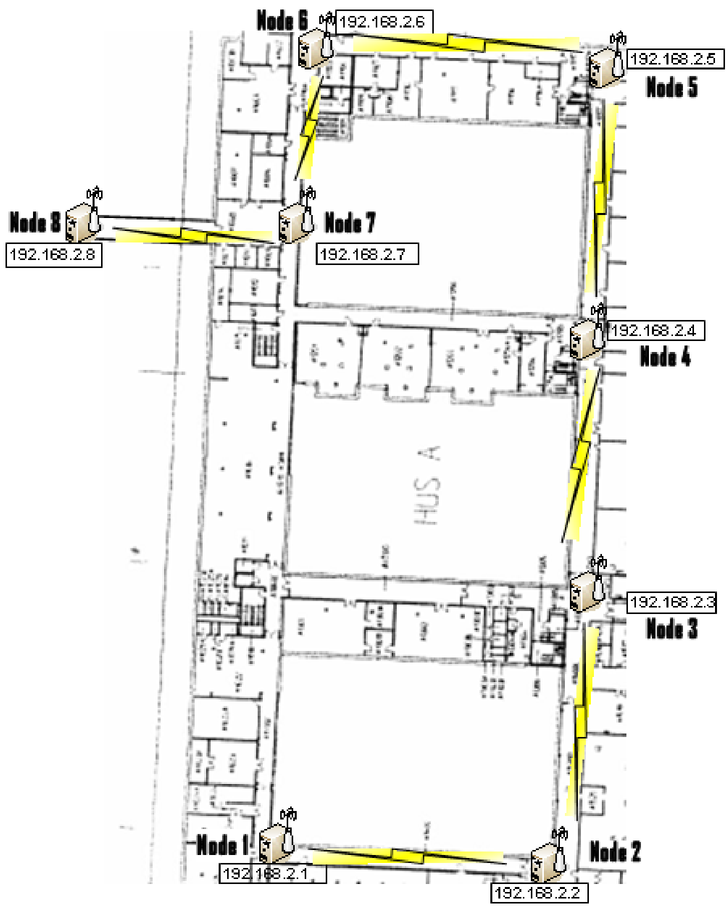

Figure 2.6: Topology experiment II of the WMN test-bed

With this topology we wanted to obtain less interferences between nodes as possible and also that communications between nodes flowed to the nearest node. The distances between nodes are described in *Table 2.5*. To be sure that communications flowed to the nearest node we used static routing by IP and by MAC address filtering. The parameters of this network topology are defined in *Table 2.6* where we can see that we have changed some parameters from the topology of the experiment I. These changes were done because we wanted to set parameters right, learning from the experiment I and not falling in the

same errors. We have changed the transmission channel to 3 because we saw that it was the least used channel. To generate TCP traffic we set some parameters with *Iperf*. All these parameters are described in *Table 2.7*.

	Distances
Node 1 → Node 2	37.96 m
Node 2 → Node 3	36.65 m
Node 3 → Node 4	36.65 m
Node 4 → Node 5	33.79 m
Node 5 → Node 6	39.21 m
Node 6 → Node 7	25.41 m
Node 7 → Node 8	44.10 m

Table 2.5: Distances between nodes on the topology of experiment I

Parameter	Value
Routing	Static (IP and Mac)
Data rate	11 Mb/s
Transmitted signal power	18 dBm
Channel frequency	2.422 e6 (channel 3)
RTS/CTS	Off
Fragmentation threshold	Off
Power management	On

Table 2.6: Network parameters of experiment II

Parameter	Value
Length of buffer to read or write	8 KB
Server port to listen	5001
TCP window size (socket buffer size)	46.72 KB (32*MSS)
TCP maximum segment size (MTU - 40 bytes)	1.460 KB

Table 2.7: Iperf TCP settings

Scenarios

In this experiment we wanted to do a deep analysis, focusing on knowing the performance of this node when communicating with the others and how the networks worked with different numbers of nodes involved in the test.

In the scenario 1 (*Figure 2.7*), we wanted to know the performance between nearest nodes (one hop). What we did was run the client on the first node and the server on the second and sent traffic between them, and after finishing it, run the client on the second node and the sever on the third node and so on until the last node.

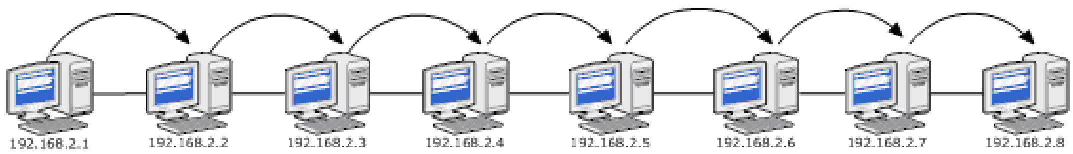

Figure 2.7: Network scenario 1

In the scenario 2 (*Figure 2.8*), we wanted to test the performance of data traffic between 3 nodes (two hops). We run the client in one node and we run the server on the second nearest node, repeating it until the last node. With this scenario we got 6 different samples.

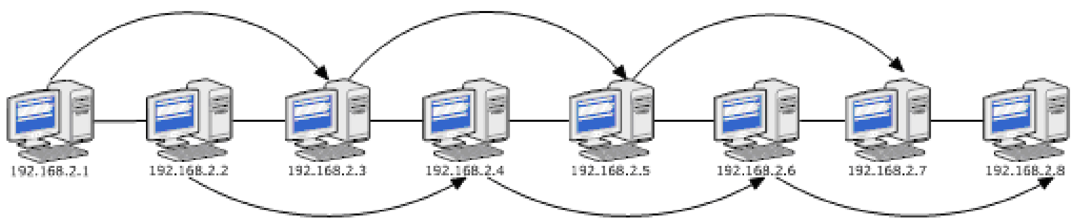

Figure 2.8: Network scenario 2

In scenario 3 (*Figure 2.9*), the traffic flowed between the first node and the fourth node (three hops) and so on until the last node. With this scenario we got five different samples of the network with which we could compare to know if the performance was similar.

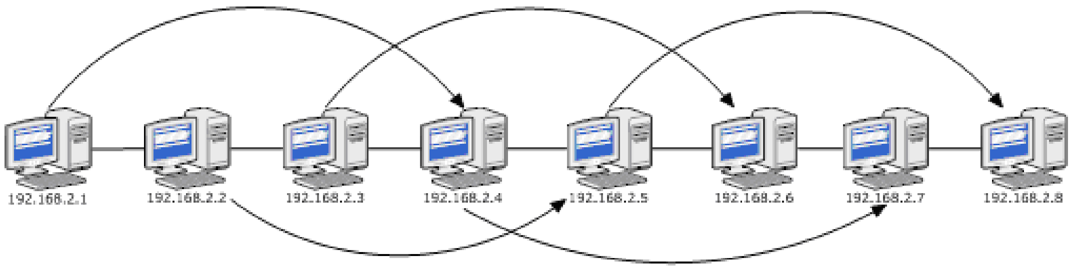

Figure 2.9: Network scenario 3

Scenario 4 (*Figure 2.10*) tested the performance of having traffic between five nodes (four hops) getting four different samples in four different places of the nodes.

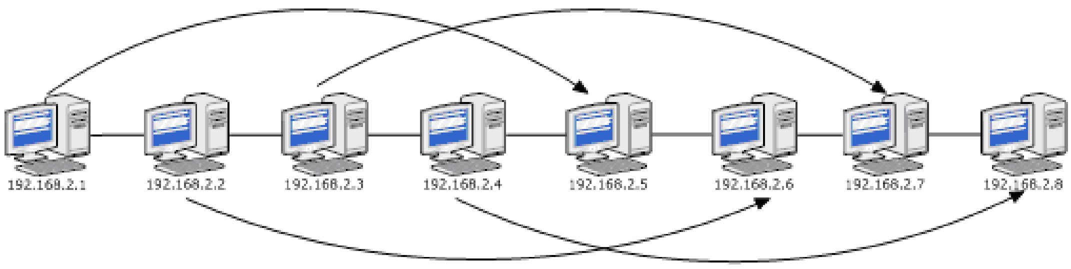

Figure 2.10: Network scenario 4

On scenario 5 (*Figure 2.11*) we obtained the performance of communication between six nodes (five hops), having three different samples.

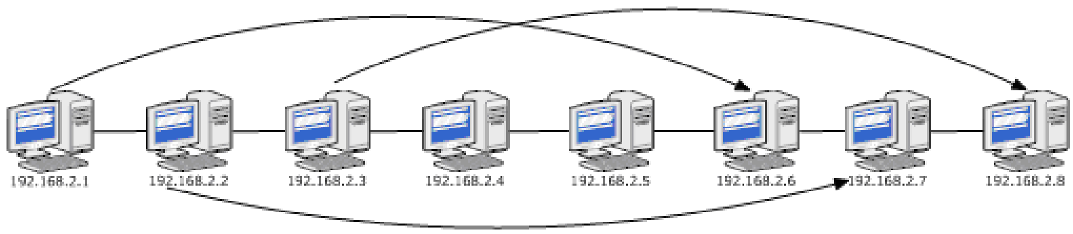

Figure 2.11: Network scenario 5

With scenario 6 (*Figure 2.12*), we wanted to test the performance of of send traffic in the WMN with six hops.

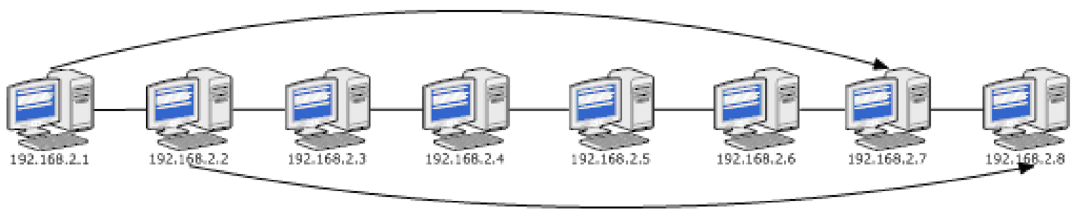

Figure 2.12: Network scenario 6

Scenario 7 (*Figure 2.13*) tested the test-bed with 7 hops, obtaining performance of the network when the traffic flows from the first node to the last one.

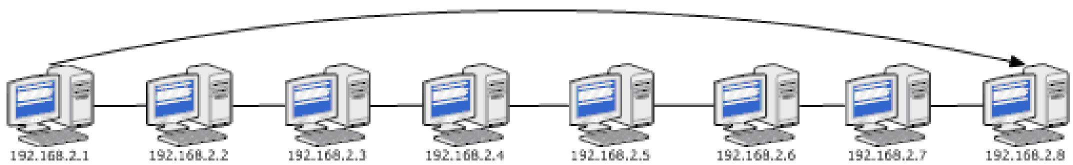

Figure 2.13: Network scenario 7

For this experiment, we determined the number of repetitions, which communication protocol to use, and how much time takes each test sending traffic. We also decided the number of runs that we were going to do in each scenario and the time taken by each trial (see *Table 2.8*).

Scenario	Number of runs	Duration (min)
1	42	126
2	36	108
3	30	90
4	24	72
5	18	54
6	12	36
7	6	18
Total(theoretic): 504 min = 8,4 h		
Total(practical): 7 am - 1.35 am = 18,59h		

Table 2.8: Real and expected time for the test-bed experiment II

When we run this experiment, we were helped by the Communication Network Research Group with the permissions needed to put all the computers in the corridors, helping us putting the computers working and also developing a script to run the different scenarios. The script was developed by the PhD student Anna Chaltseva.

2.5.4 Results of experiment II

We measured the available throughput in terms of sent pay-load at Layer 4 with IP/TCP as bearer in the scenario of a single client. To obtain the results of this experiment, we have used *tcpdump* to capture all the traffic generated in the network. When each trial finishes, *tcpdump* creates a *".pcap"* file that contains all the information about what happened in the network while we were testing it. Using *Wireshark* we calculated the throughput of each trial. We can see in *Appendix D* all the tables with all values calculated.

Hop by hop

The first analysis of the network we wanted to do was the performance of the network when we sent traffic from one node to another with different hops between them. In the following figures we show the trials (X axis) and the throughput (Y axis) and we plot the traffic flow between the different nodes.

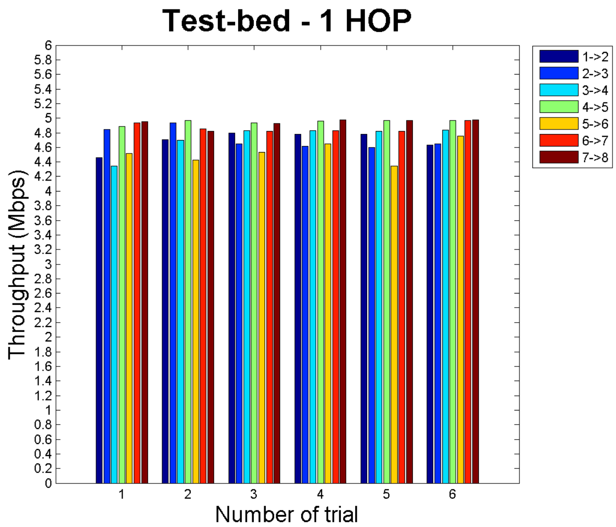

Figure 2.14: Average throughput test-bed for 1 hop

In *Figure 2.14* we can see the behaviour when the transmission is between two neighbouring nodes. We see that the worst throughput is from 5 to 6, but the difference between this performance and the other nodes does not show any anomalous interference.

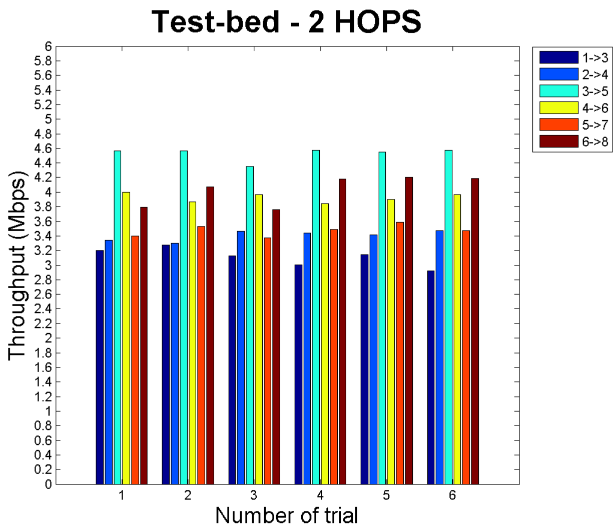

Figure 2.15: Average throughput test-bed for 2 hops

In *Figure 2.15* we see that the difference between the behaviour of sending from 1 to 3 and sending from 3 to 5 is a big difference (more or less 2.4 Mbps). This is because if we see where they are placed, we see that to go from 1 to 3, packets have to cross a corner. We have the same problem with the 5 to 7 transmission, but the throughput is higher because the distance between them is smaller. Also we can see that from 2 to 4 the throughput is also very low. We can explain this because the corridor got narrower, and the node 4 received a weakness signal.

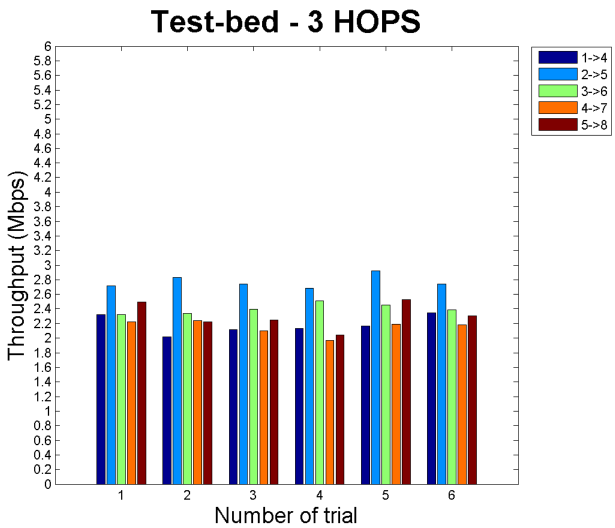

Figure 2.16: Average throughput test-bed for 3 hops

Analyzing the performance when we send traffic through 4 nodes (*Figure 2.16*), we can realize that the pattern noticed in the two hops graphic is also reference here. We can see how the lowest throughputs are when we are sending traffic from 1 to 4, from 4 to 7 and from 5 to 8, just where the transmission interferences are.

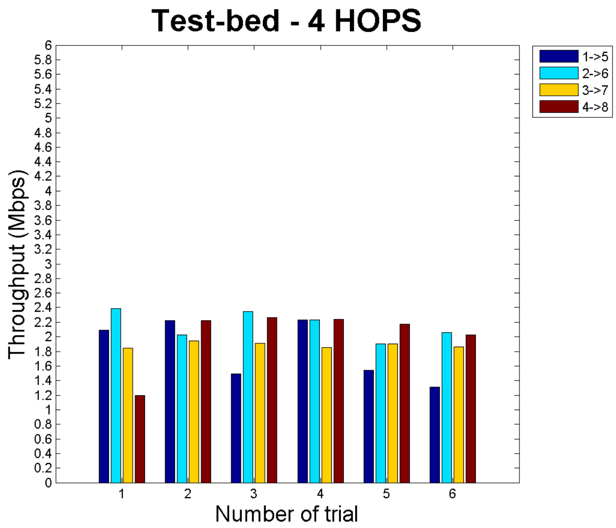

Figure 2.17: Average throughput test-bed for 4 hops

In *Figure 2.17* we see that when we were sending traffic from node 1 to 5, there are some peaks, but we continue seeing that there is an interference problem between node 1 and node 3. We notice how other three transmissions between nodes that have same the problems we notice above, the throughput is similar.

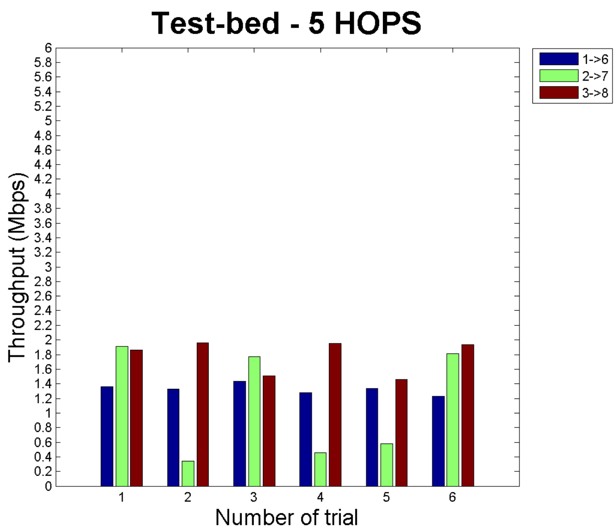

Figure 2.18: Average throughput test-bed for 5 hops

We can see in *Figure 2.18* that the throughput when transmitting from node 2 to 7 varies a lot, this is due to other people's interferences because when we were testing this scenario we detected with *Network Stumbler* that other people were transmitting on the same channel (channel 3), which made more interferences for our transmission. At first, when we detected this problem, we thought that it was a good idea to change our transmission channel to one not so busy, but in the end we discarded this idea because we realized that other users transmissions were lower and because all other channels we analysed were also very busy.

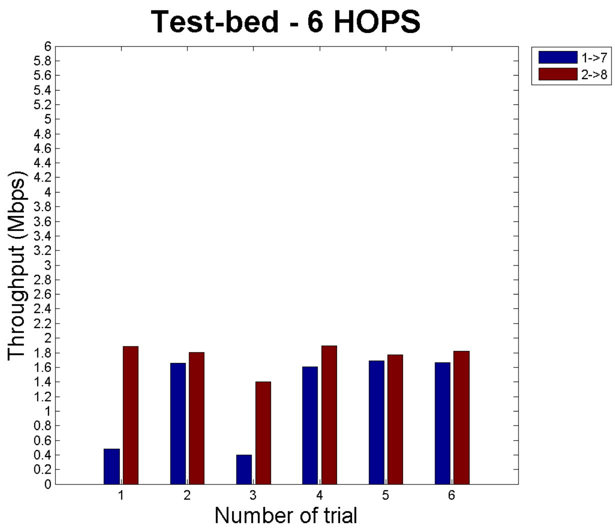

Figure 2.19: Average throughput test-bed for 6 hops

When we started with the scenario number six (*Figure 2.19*), we kept having some interference problems related with other people transmissions in the same channel, but we found who was transmitting and asked him to stop while we were doing the test, that is why the throughput stabilized. We can also see that this stabilization continues when we tested the 7 hops graphic (*Figure 2.20*).

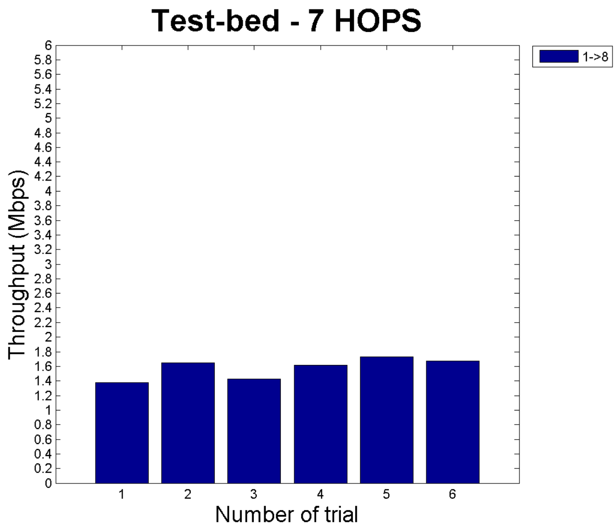

Figure 2.20: Average throughput test-bed for 7 hops

It is shown in all the graphics above that there are reasonable interferences according to corridors characteristics and people walking around the building.

Total throughput

In *Figure 2.21* we can observe the throughput average for different hops. We can see how the throughput between two nodes has not equal decrease when the number of hops increases. This is a natural behaviour that can be explained from different aspects:

- We cannot achieve a throughput equal to the channel data rate (11 Mbps) because in a wireless channel the signal sent from the source looses its strength at the receiving end because of path loss (multipath propagation). That is why the more nodes the packets go through the more the throughput decreases.

- When transmitting a TCP packet, the acknowledgment is sent out after the transmission has finished and certain duration of time called short inter frame spacing (SIFS) has elapsed. If a node wants to transmit a frame and senses the medium idle for a certain duration of time called distributed coordination function inter frame spacing (DIFS), it may start transmitting [21]. The sum of these times not used for transmission produces the decreasing of the throughput.

- In the indoor propagation case the building material used in floor, ceiling and walls also affects the signal strength, and the throughput is directly proportional to the received signal strength [22].

- We have performed our experiments in the ISM 2.4GHz band, so there are also co-channel and overlapping channel interferences to our used channel of transmission (channel number 3). The co-channel and overlapping channel interferences further weakens the received signal strength.

Due to these reasons we received a throughput less than the channel data rate.

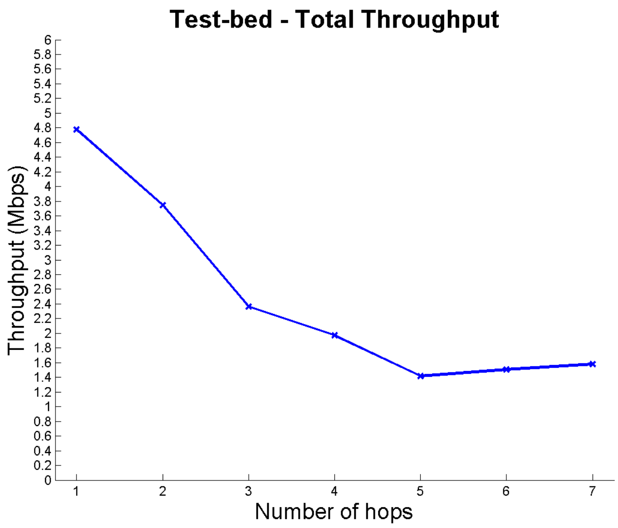

Figure 2.21: Test-bed total throughput

We also can see in *Figure 2.21* that when the throughput increases in 6 and 7 hops progressively, that is not normal but it has an explanation. When we were testing the network we found a problem, mentioned above, related to transmission interferences. To try to solve it, apart from thinking about changing our transmission channel, we moved some antennas from some nodes. Doing it, we improve the signal strength between some nodes which produced an increasing of the throughput.

If we pay attention to confidence interval (*Figure 2.22*) we can see that intervals for 1, 2, 3 and 4 hops are very small, which means that the throughput in these hops is not very variable and if we run the test in the same conditions, we will get a value in this range with a probability of 90%, but we can also see that the minimum and maximum intervals in all the hops are very big compared with the confidence interval, and we can conclude that it is because there are a lot of interferences that produced some peaks in the transmission.

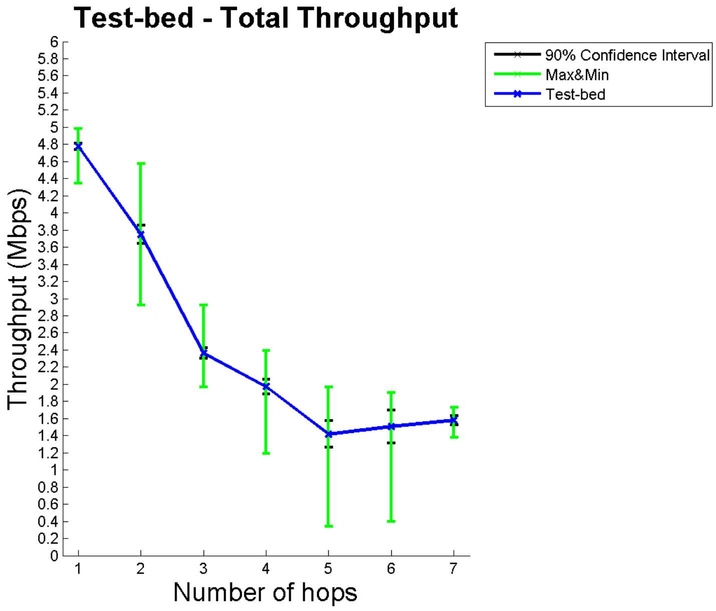

Figure 2.22: Test-bed total throughput vs. 90% confidence interval vs. maximum&minimum

CHAPTER 3

WMN in the *NS-3* Simulator

This section will cover the second part of the master project. This part consists of a description of how we have used the *NS-3* Network Simulator and how we have developed a program (and seven scripts to run this program) to implement the network with the same behaviour and parameters as the network created in the first part of the thesis.

3.1 Understanding the *NS-3* Network Simulator

The *NS-3* Network Simulator is a discrete-event network simulator with a special focus on Internet-based systems. *NS-3* is a user-space program that runs on UNIX and Linux-based systems and on Windows. We install it on Ubuntu as in the test-bed and we used 3.7 version, the latest stable release of this network simulator.

In *NS-3*, simulation or library components are written in C++, with has support for extensions that allow simulation programs to be written in Python. These Python bindings should be written, but at the end the objective is having an API support at the Python level.

3.1.1 Scope of *NS-3*

The focus of *NS-3* is on IPv4 and IPv6-based networks, but also other architectures such as sensors or DTNs are to be supported. *NS-3* is meant to be modifiable and extendable by users. Some users are able to use example programs that are provided, but it is expected that users will want to write new programs, and modify or add models to the simulator in some way. Source code distributions are therefore expected to be the preferred means for distributing *NS-3*.

3.1.2 Supported mechanisms and protocols

NS-3 has a modular implementation containing different libraries supporting the simulator (and it is also possible that users can write and link their own libraries):

- **Core library:** Offers support for generic aspects of the simulator, such as to generate random numbers, use smart pointers, callbacks, or debugging objects.
- **Simulator library:** Defines simulation parameters such as simulation time, objects, schedulers, and events.
- **Common library:** Defines independent objects such as generic packets and tracing objects.
- **Node library:** Defines abstract classes for fundamental objects in the simulator, such as nodes, channels and network devices.
- **Internet-node:** Defines internet-related models such as TCP/UDP protocols.

The modular implementation allows smaller compilation units and when compiling is only necessary to compile the program changed. *NS-3* executable programs may be built to either statically or dynamically link the libraries. *NS-3* offers support for the following:

- Construction of virtual networks (nodes, channels, applications) and support for items such as event schedulers, topology generators, timers, random variables, and other objects to support discrete-event network simulation focused on Internet-based and possibly other packet network systems.
- Support for network emulation: the ability for simulator processes to emit and consume real network packets.
- Distributed simulation support: the ability for simulations to be distributed across multiple processors or machines.
- Support for animation of network simulations.
- Support for tracing, logging, and computing statistics on the simulation output.

3.1.3 Use cases

To use *NS-3* we have to know how it is designed. For that, we describe in this section design issues and usage models, and mention trends in simulation use within the networking research community.

Model extensibility

Users are interesting in extending the simulator by writing or modifying simulation programs. To make it possible, *NS-3* uses object-oriented design with polymorphic classes, allowing users to modify the aspects they want to change. To facilitate the addition of new models, *NS-3* adopts a component-based architecture for compile-time or run-time addition of new models, interface aggregation, and encapsulation, without requiring modification of the base models of *NS-3*.

Run-time configuration

NS-3 allows users to redefine default values and class types without recompiling the simulator. The default database values are integrated with a command-line argument parsing facility, making all the variables configurable from the command-line as well.

Tracing

NS-3 features a callback-based approach to tracing that difference between sources and destination. Packet traces are available in *"pcap"* format, to allow analyzing these files using network protocol analyzers such as *Wireshark* or *Tcpdump*.

Scaling

It is scheduled that *NS-3* will include techniques for improving the scalability of simulations, including distributed simulation techniques introduced with PDNS and GTNetS, scalability techniques introduced for wireless simulations such as caching of computationally-intensive results, and fexibility in tracing infrastructure (to avoid large traces).

Software integration

NS-3 is oriented to reuse existing software such as *NS-2* programs or other applications. The *NS-3* design is built around encapsulation techniques separating the application interface from the implementation. *NS-3* has also a library that allows implementation code to run in both real and simulated environments.

Network emulation

The *NS-3* design is intended to facilitate interaction between simulation and experiments, with encapsulation techniques that allows real application and kernel code to run in the simulator, thereby improving traceability to real-world implementations.

Programming

The *NS-3* user interface at present is a C++ "main" program. However, *NS-3* will also feature Python bindings allowing users to define programs and replaceable components in Python.

3.1.4 Key simulation objects

This section walks through the primary simulation objects in the simulator, related to the sending and receiving of packets between nodes. In *Figure 3.1* [8], it is shown graphically the relations between objects.

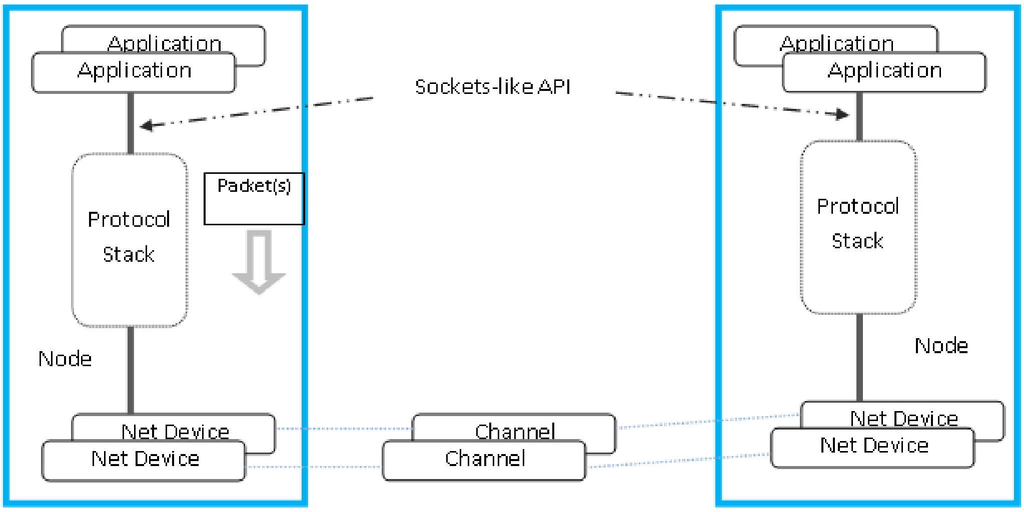

Figure 3.1: NS-3 *key simulation objects architecture*

Node

Class *Node* is intended mainly as a base class in *NS-3*, but it can be instantiated as well (i.e., it is not an abstract class). Users can create their own *Node* subclasses, and *NS-3* will provide a few.

The design uses patterns of software encapsulation to allow *Applications* and *NetDevices* (other class explained bellow) to talk to implementation independent interfaces of the underlying TCP/IP implementations.

NetDevice and Channel

Class *NetDevice* represents a physical interface on a node (such as an Ethernet interface). The basic idea is to simulate the Linux architecture at the boundary between the device-independent sub layer of the network device layer and the IP layer.

Class *Channel*, which is closely coupled to the attached *NetDevices*, implements a logical path over which the information flows.

Packet

NS-3 Packet objects contain a buffer of bytes: protocol headers and trailers are serialized in this buffer of bytes using user-provided serialization and deserialization routines. The content of this byte buffer is expected to match bit-by-bit the content of a real packet on a real network implementing the protocol of interest.

The design of the Packet framework of *NS-3* was heavily guided by a few important use-cases:

- Avoid changing the core of the simulator to introduce new types of packet headers or trailers.
- Maximize the ease of integration with real-world code and systems.
- Make it easy to support fragmentation, defragmentation, and, concatenation which are important, especially in wireless systems. It is quite natural to implement with this packet design, since we have a buffer of real bytes, we can split it in multiple fragments and reassemble these fragments.
- Make memory management of this object efficient.
- Allow actual application data or dummy application bytes for emulated applications.

Applications

Applications are user-defined processes that generate traffic to send across the networks to be simulated. *NS-3* provides a framework for developing different types of applications that have different traffic patterns. There is an *Application* base class that allows one to define new traffic generation patterns via inheritance from this class. Then one simply creates the application and associates it with a node, and the application will send traffic down the protocol stack. Applications on a node communicate with the node's protocol stack via sockets.

3.2 *NS-3* Test-bed implementation

In this section we present the decisions we have taken when implementing our test-bed in *NS-3* simulator, and describe the problems we have found and what we have done to solve them.

3.2.1 Topology

The topology we implemented was the topology of the test-bed of experiment II (see section 2.5.3). We focused on simulating that topology and setting all the parameters according to the concrete characteristics of the place and all the interferences it produces to the network.

As we commented in the last chapter, it was a chain topology network so we put the nodes in a line, setting the real distance between them (see *Table 5*) as it is shown in *Figure 3.2*.

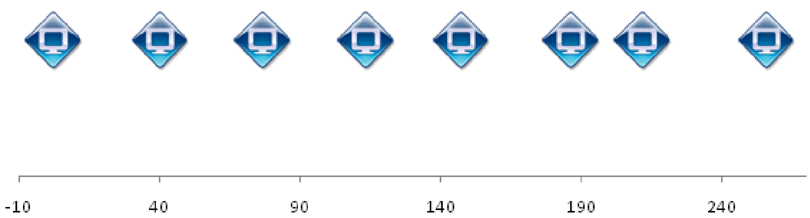

Figure 3.2: Coordinate system of the NS-3 *chain topology*

3.2.2 Propagation loss model

One of the most important parameters when we talk about wireless networks are the interferences (such as electrical equipment, other wireless networks, etc.) and obstacles (as walls, ceilings, doors, people moving, etc.) present in a real-world transmission medium. Due to all these interferences, the signal strength decreases in different ways. In order to set the test-bed interferences behaviour in *NS-3*, we have collected from the test-bed the average of the signal strength received by each node from the other nodes while we were doing the experiments and, with this information and knowing the distance between nodes we found a *NS-3* propagation loss model according to this information. In *NS-3* there are different propagation loss models (see *Figure 3.3*). We reviewed all of them to see which one adapted better to our test-bed requirements.

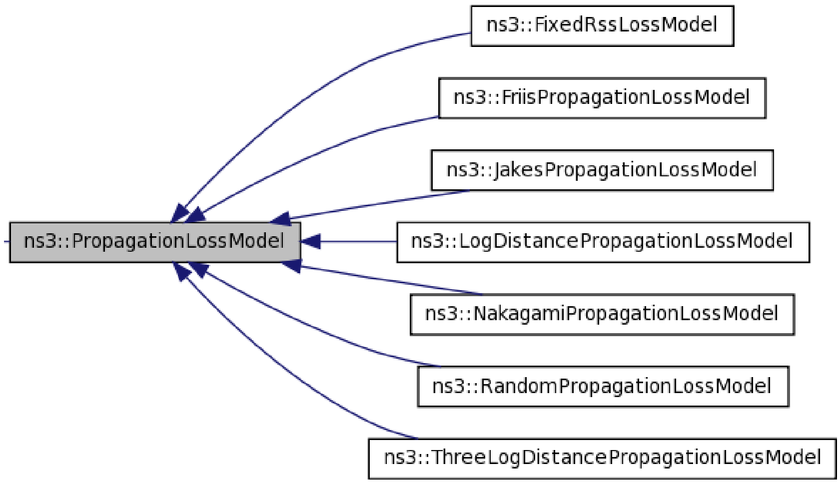

Figure 3.3: Different NS-3 *propagation loss model*

Fixed Rss Loss Model

With this model we can fix the propagation loss by setting the received power level (RSS, measured in dBm). We dismissed this model because it does not take the distance between nodes into account.

Friis Propagation Loss Model

This model uses an equation that gives the power received by one antenna under idealized conditions given the another antenna some distance away when transmitting a known amount of power. This model was interesting for us because with it we could calculate the power received.

Jakes Propagation Loss Model

This model allows setting some physical parameters such as:

- The number of rays used by default to compute the fading coefficients for a given path.
- The number of oscillators used by default to compute the coefficient for a given ray of a given path.
- The Doppler frequency in Hz.
- The distribution to choose the initial phases.

We did not have these parameters from our test-bed, that is why we dismissed this model.

Nakagami Propagation Loss Model

This model is used to describe the statistical properties of a wireless channel since the signal propagation is affected by three statistically independent phenomena: deterministic path loss, slow lognormal shadowing and fast multipath fading. We dismissed it because it used as a reference two distances and it does not take the strength of the signal received into account.

Random Propagation Loss Model

This model is used when a parameter is included in some specifics values, i.e. distance. We did not use it for the distance because our nodes were static, but we used it to introduce a variance in the signal strength (it is explained in this chapter how and why we used it).

Three Log Distance Propagation Loss Model

This model is equal to the *Log Distance Propagation Loss Model*, but this model used three distance fields instead of one. We only needed one distance field, that is why for us it was better to use the *Log Distance Propagation Loss Model*.

Log Distance Propagation Loss Model

This model calculates the reception power with a so-called log-distance propagation model. We used this model in our simulation program because it was the one that better adapts to our network model. This model calculates the reception power (received signal strength) using the following equation:

$$L = L_0 + 10 \cdot n \cdot \log_{10} \frac{d}{d_0}$$

Where:

- L_0 **:** Reference distance (m)
- $\boldsymbol{n}$ **:** The path loss distance exponent
- $\boldsymbol{d}$ **:** Distance (m)
- d_0 **:** Reference distance (m)
- $\boldsymbol{L}$ **:** Path loss (dB)

To use this model for the test-bed we have had to estimate or calculate the value of these variables to adapt it to the real test-bed.

Propagation Loss Model Used

We used three propagation loss models and one propagation delay model in the *NS-3* implementation: *Log Distance Propagation Loss Model*, *Friis Propagation Loss Model*, and *Random Propagation Loss Model*.

We used the *Friis Propagation Loss Model* knowing that this loss model is valid in the far field of the antenna, where far field starts far the beyond the *Rayleigh distance*,

$$RayleighDistance = \frac{2 \cdot L_a^2 \cdot f}{c}$$

Where:

- L_a **:** Antenna size (in our case 1)
- **f :** Transmission channel frequency
- **c :** Speed of light ($3 \cdot 10^8$ m/s).

In our case, *Rayleigh distance* is:

$$RayleighDistance = 16.14\hat{6}$$

That is, the transmitter and receiver should be at a distance greater than the *Rayleigh distance* to calculate L_0 (as we can see in *Table 7* all distances between nodes are bigger than the *Rayleigh distance*),

$$L_0 = 20 \cdot \log_{10} \frac{4 \cdot \pi \cdot d_0 \cdot f}{c}$$

Where:

- d_0 **:** Reference distance (1m in our case)
- $\boldsymbol{f}$ **:** Transmission channel frequency (in our case, Channel 3 = $2.422 \cdot 10^9$ Hz)
- $\boldsymbol{c}$ **:** Speed of light ($3 \cdot 10^8$ m/s).

In our case L_0 is:

$$L_0 = 40.125$$

Now that we have calculated the value of L_0, we need to estimate the n value to apply the *Log Distance Propagation Loss Model*. To estimate n, the path loss distance exponent, we now that the test-bed testing was done indoor and we based our estimation on measurements acquired in a experiment for a WLAN indoor office environment [23],

where it is said that "The indoor channel study requires *N=18* for a LOS path between the transmitter and the receiver (a path loss exponent equal to 1.8)"; because of that, we set n to 1.8.

To model exactly the values we obtained from the test-bed we added to this model the *Random Propagation Loss Model* to make it work in the range of signal strength where the test-bed works. The ranges of the received signal strength in the test-bed are values between -67 and -83 dBm, that is why we add a random model working between these parameters, setting the global received signal strength of the network in the range (L_0+27, L_0+43).

We also added a *propagation delay model* due to the interferences of the air, setting it according with the speed of light ($3 \cdot 10^8$ m/s).

3.2.3 Implementation

In this section it is described how the test-bed simulation program has been developed, the classes used and the problems founded developing it. In *Appendix C* it is shown the program code, how to run it and, it is also explained the class developed (*MeshTest*) and all the attributes and methods it use.

Problems with 802.11s implementation

When we set the protocol to use in the program of the simulation, we started using the *NS-3* 802.11s implementation [24]. We used the class *MeshHelper* to install it.

For the first trials we sent TCP traffic from one node to another and it worked perfectly. After checking that everything worked correctly, we introduce the Propagation loss model mentioned above. When we tried to run it with the new features, it did not send any packets. To check that the problem was with the 802.11s implementation we changed to the IEEE 802.11b protocol, and we run the program using this protocol and the Propagation loss model and it worked perfectly. With this verification we concluded that there were some problems with the 802.11s implementation.

We sent an email to the *NS-3* developers reporting the problem but we did not get any answer. Due to this problem we decided to simulate the test-bed using the protocol IEEE 802.11b. This is possible because on the real test-bed we did not use any of the features that the IEEE 802.11s protocol offers, but anyway we were testing the 802.11s implementation without using some features, so all the settings on the test-bed can be setting up using IEEE 802.11b (we are also using 11 Mbps data rate).

Nodes

We started creating the nodes using the class *nodeContainer* that contains 8 nodes. To set the parameters of the WiFi-card we used an object of the *NetDeviceContainer* class, and to set the interfaces of each node we used an object of the *Ipv4InterfaceContainer* because we were using the Internet Protocol version 4. After we set all the parameters of each object, we set the position of each node with the *MobilityHelper* class and we installed on nodes the devices and interfaces.

Devices

In the devices we set the Mac layer using the *NqosWifiMacHelper* class setting ADHOC as the routing protocol . We used the *YansWifiChannelHelper* class to set the Propagation loss model described above. To set up physical settings we used the *YansWifiPhyHelper* class, and we set some parameters such as the *Energy Detection Threshold*, the *Transmission Gain*, the *Transmission Power*, the *Channel Number*, etc. according with the test-bed values.

Interfaces

Using the *InternetStackHelper* class, we set the internet stack and for assigning IP address to each interface we use *Ipv4AddressHelper*. To set the static routing we used *Ipv4StaticRoutingHelper*.

Application

To install applications in the nodes we used *ApplicationContainer* and for sending traffic between nodes we use *PacketSinkHelper* which installs a TCP traffic generator on the client(*TcpSocketFactory* class).

Random Runs

We used the class *seedManager* to generate random executions, with a seed setting with a default value, but with the option of changing it as input of the program by value.

3.3 NS-3 Experiments

Once we developed the program, we wanted to run the same experiment as run in the test-bed, described in section 2.5.3. Theoretic times expected in these experiments were the same as for experiment II, but the real time changed (see *Table 3.1*). To run this experiment we developed some scripts that run all the scenarios (see *Appendix C*), six trials per scenario. As *NS-3* is a simulator we changed the seed of the random generator each time we ran the program to obtain different simulations.

Scenario	Number of runs	Duration (min)
1	42	84
2	36	72
3	30	60
4	24	48
5	18	36
6	12	24
7	6	12
Totally (theoretic): 336 min = 5,6 h		
Totally (practical): 13385 s = 3,7h		

Table 3.1: Real and expected time for simulation

3.4 *NS-3* Results

To obtain the results from the simulation, after each run the program generated a *".pcap"* file for each node, which contained all the packets captured. We focused our interest in analyzing the client node (node generating TCP traffic), then once we got the file we analyzed it using *Wireshark* in the same way as done in test-bed experiment II (in *Appendix D* all the throughput values for this experiment are shown).

Hop by hop

As we did with the test-bed results, the first analysis of the throughput we wanted to do was to know the behaviour of the network hop by hop.

Observing *Figure 3.4* we can notice that there is a big difference between the throughput obtained when we send traffic from node 6 to 7 (that is very high) and the throughput between node 7 and 8. This behaviour is due to the Propagation loss model that we were using, establishing the distance as principal parameter. If we look at Table 2.5, and if we order the pairs of nodes by increasing distance we obtain this list: $6 \rightarrow 7$, $4 \rightarrow 5$, $3 \rightarrow 4$, $2 \rightarrow 3$, $1 \rightarrow 2$, $5 \rightarrow 6$, $7 \rightarrow 8$. As we can see, it is just the same order if we compare from best throughput order.

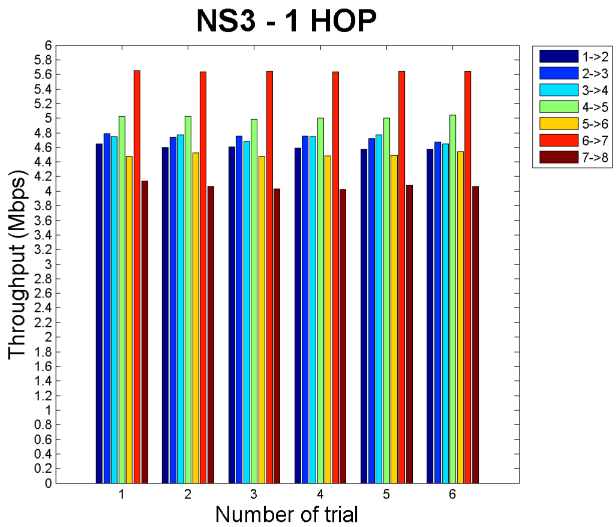

Figure 3.4: Average throughput of NS-3 *for 1 hop*

With two hops we can see how distances were similar (it was the sum of the distances between the node in the middle with the others two nodes in each interval, *Figure 3.5*), but we can notice that a difference received signal strength influence on the traffic sent between nodes.

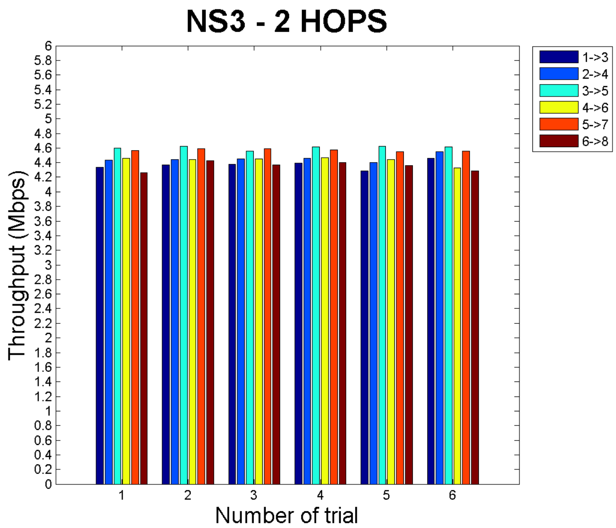

Figure 3.5: Average throughput of NS-3 *for 2 hops*

In *Figure 3.6* we can observe when we send from node 5 to 8 how distance between nodes there keeps being a serious interference when distances are not similar.

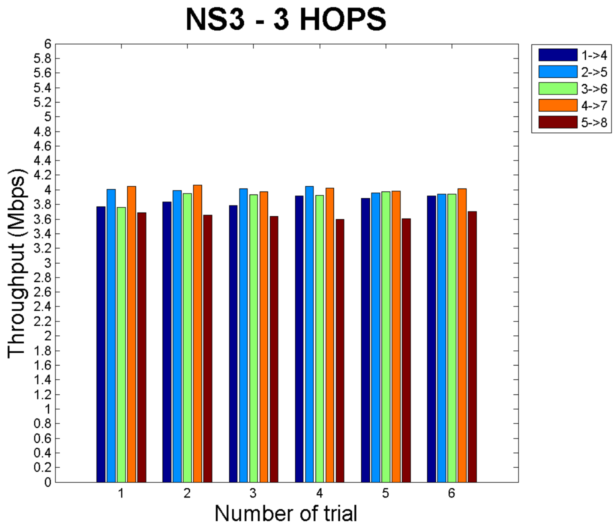

Figure 3.6: Average throughput of NS-3 *for 3 hops*

Looking at *Figure 3.7*, we can observe also some peaks due to the propagation delay model according with indoor conditions.

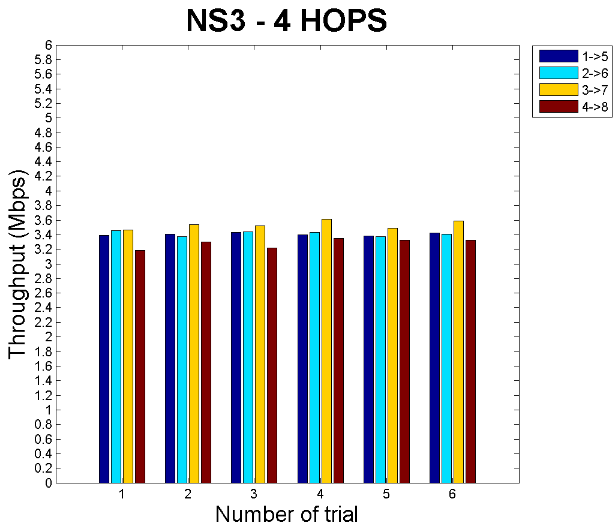

Figure 3.7: Average throughput of NS-3 *for 4 hops*

In *Figure 3.8*, *Figure 3.9* and *Figure 3.10* we can observe the same behaviour detected in the other coordinate systems.

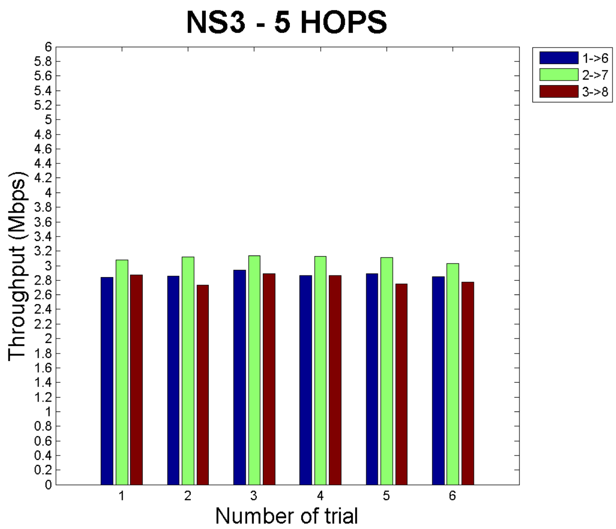

Figure 3.8: Average throughput of NS-3 *for 5 hops*

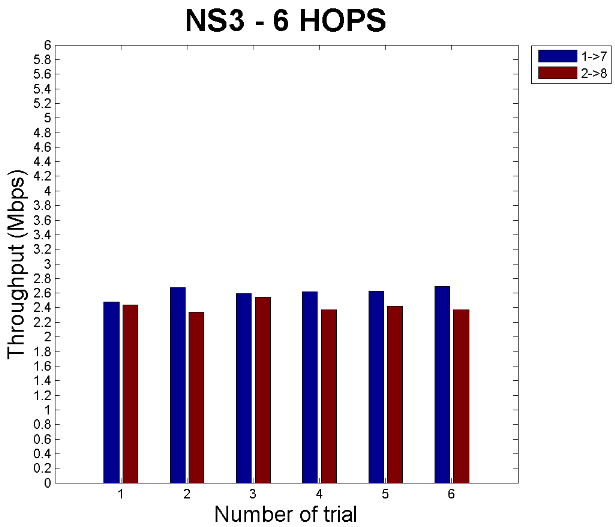

Figure 3.9: Average throughput of NS-3 *for 6 hops*

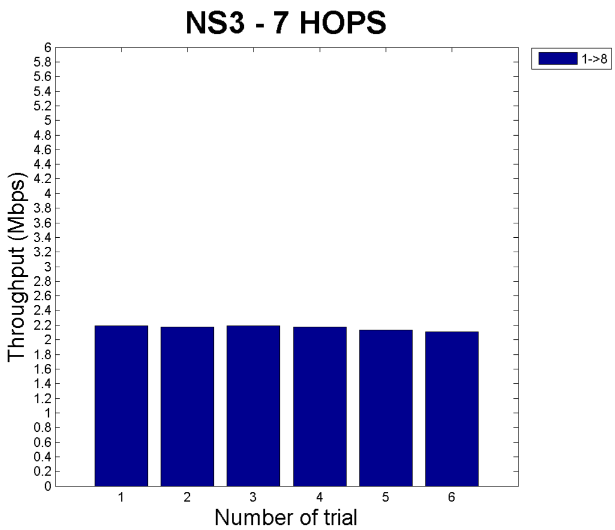

Figure 3.10: Average throughput of NS-3 *for 7 hops*

It is shown in all the graphics above that there are reasonable interferences similar to the interferences introduced in our implementation.

Total throughput

We can see in *Figure 3.11* how the throughput decreases linearly with the increase of the number of hops. We can see also that with two hops the throughput decrease less than any other time because the distances were similar. We can deduce that the throughput decrease proportionally with the distance.

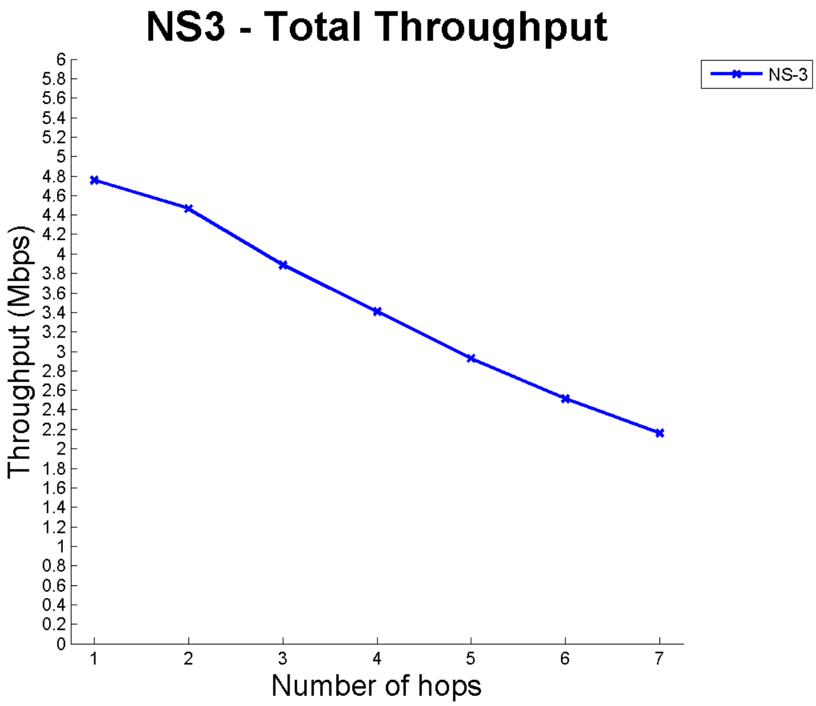

Figure 3.11: NS-3 *total throughput*

If we pay attention to the confidence interval (*Figure 3.12*) we can see that for 1 hop it is bigger than for the rest of hops. This is due to the fact that there were many different distances when using 1 hop and in the other different hops the distances are similar. Looking at maximum and minimum values, we can also see the same that for one hop the range of throughput is bigger than with the others hops.

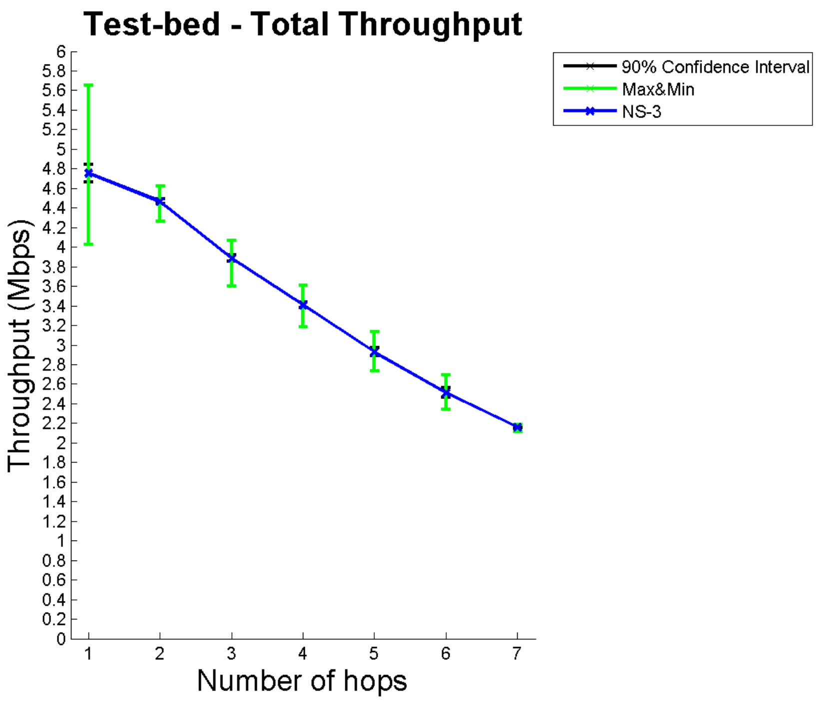

Figure 3.12: NS-3 *total throughput vs. 90% confidence interval vs. maximum&minimum*

CHAPTER 4

Comparing results

In this chapter, the results obtained from the experiments done in the test-bed and in the *NS-3* simulation are compared. We are going to do it in two different ways. First, we are going to compare them using the throughput calculated in the experiments, and finally some *".pcap"* files from the test-bed with the corresponding ones of the simulation are analyzed.

4.1 Throughput

The throughput has been analyzed from the test-bed and from *NS-3*. Now it is time to compare both results. First of all, as it is shown in *Figure 4.1* the throughput matches when there is one hop, but as we increase the number of hops, the throughput decreases in a different way.

Looking at the test-bed throughput, it is like if there were three different stages, one between hops 1 and 3, other between hops 3 and 5, and the last one between hops 5 and 7, and looking at the *NS-3* simulation, there are two different stages, one from 1 to 2 and the other from 2 to 7. From these different stages we could say that in *NS-3*, the throughput decrease is lower but constant, while the test-bed throughput decreases in a different way depending of the number of hops. This behaviour is because in the test-bed the conditions of the place change, and so the interferences produced to the transmission system are different (as shown in the coordinate system, where the are three different places where the interferences are different), and in the *NS-3* the interferences to the system keep similar during all the experiment.

According to the maximum and minimum gap shown in *Figure 4.2*, only when there is one hop, this interval is bigger in the *NS-3* simulation than in test-bed. In the other cases, in test-bed the maximum and minimum intervals are variable because of the interferences

of the medium. If we pay attention to the confidence interval, we can say that the variance of the values is much bigger in the test-bed than in *NS-3*. This confirms that the *NS-3* transmission was done in similar conditions while test-bed conditions changed continuously.

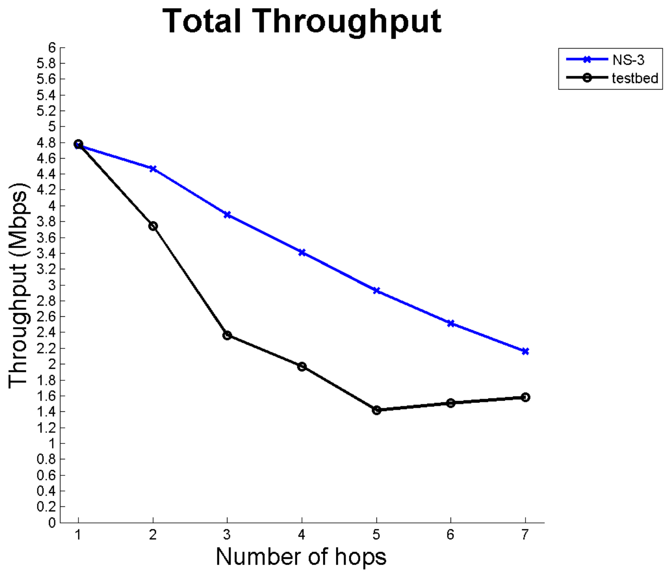

Figure 4.1: Throughput comparison between test-bed and NS-3

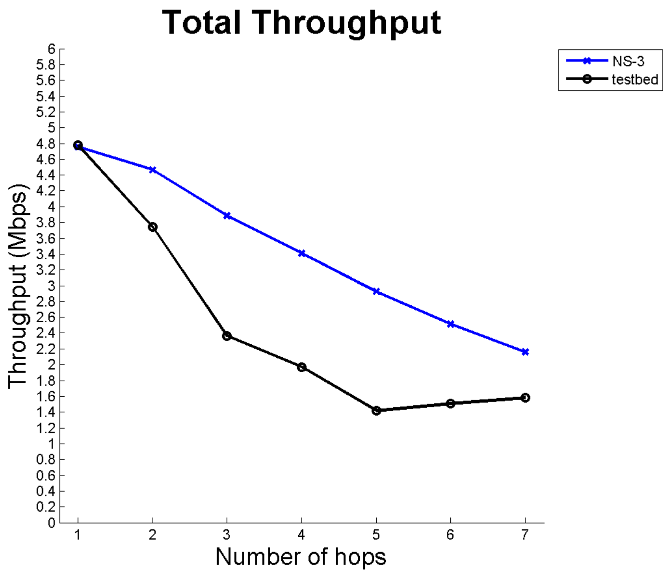

Figure 4.2: Throughput test-bed vs. NS-3 *vs. 90% confidence interval vs. maximum&minimum*

4.2 Analyzing *".pcap"* files

When the test-bed results were analysed, we found that there were big throughput differences between two different nodes with the same number of nodes between them (same number of hops). Therefore we decided to compare two of the *".pcap"* files obtained in the test-bed experiment with one of the *NS-3* simulation (only one because in this case the throughputs were similar) when they send data in two hops experiment. Specifically, we have chosen (see *Figure 2.15* and *Figure 3.5*):

- Test-bed sample 1→3: Because it was the lowest throughput where we find many interferences.
- Test-bed sample 3→5: Because it was the highest throughput.
- *NS-3* sample 1→3: Because it was the lowest throughput.

Figure 4.4, which shows measurements of round trip times of the three samples described above, illustrates that TCP adjusts its notion of average round trip time for the connection. We can see how RTT from *NS-3* is adjusted to higher values in specific moments, but the number of retransmissions is not very high. Whereas in the test-bed (specially 1→3) we can see that the number of retransmissions is higher and the RTT changes continuously due to retransmissions of different data segments. This behaviour is according to the TCP adaptive retransmission algorithm [25] that records the time at which each segment is sent and the time at which an acknowledgement arrives for the

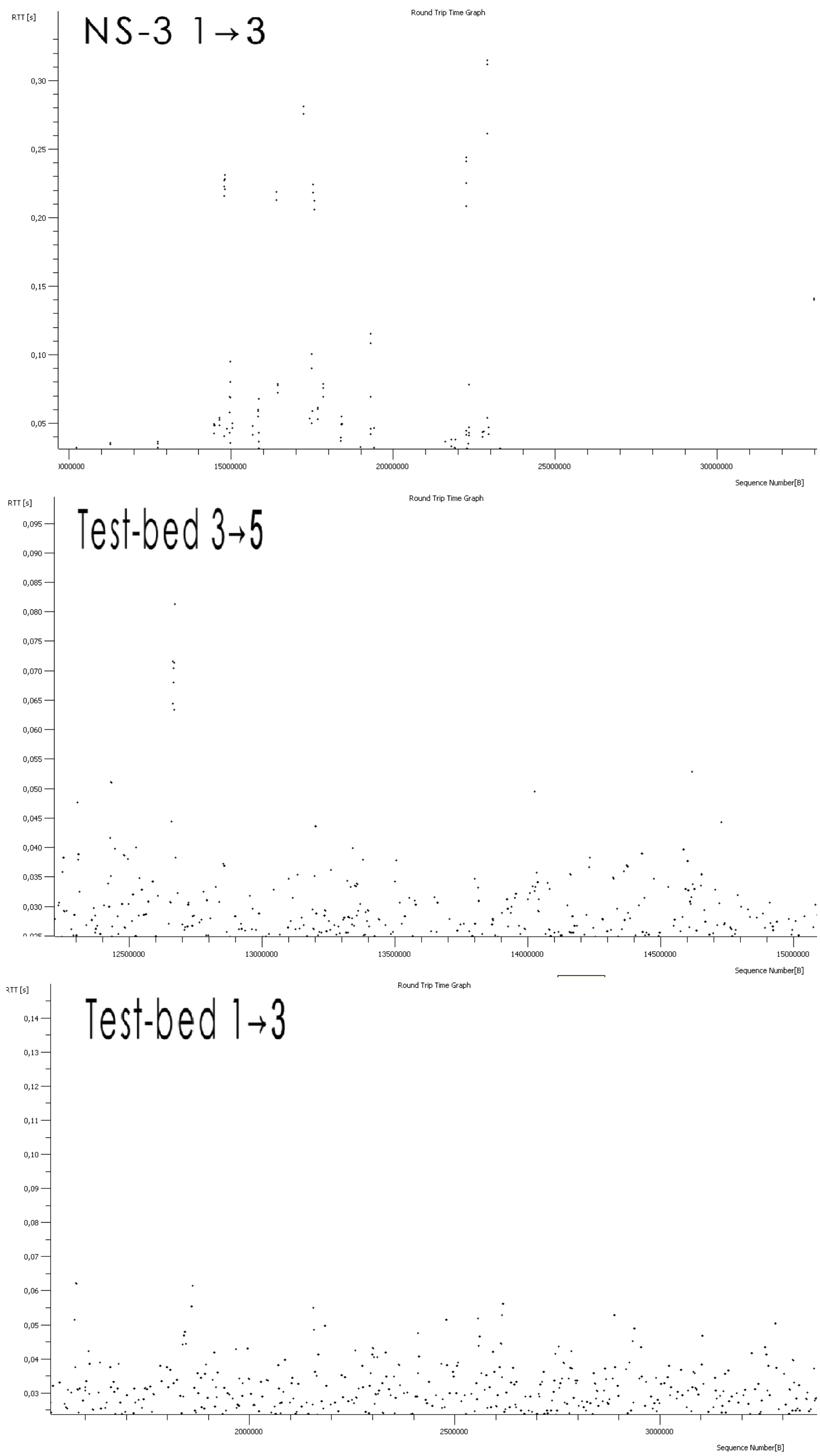

Figure 4.3: RTT from 3-samples PCAP *files*

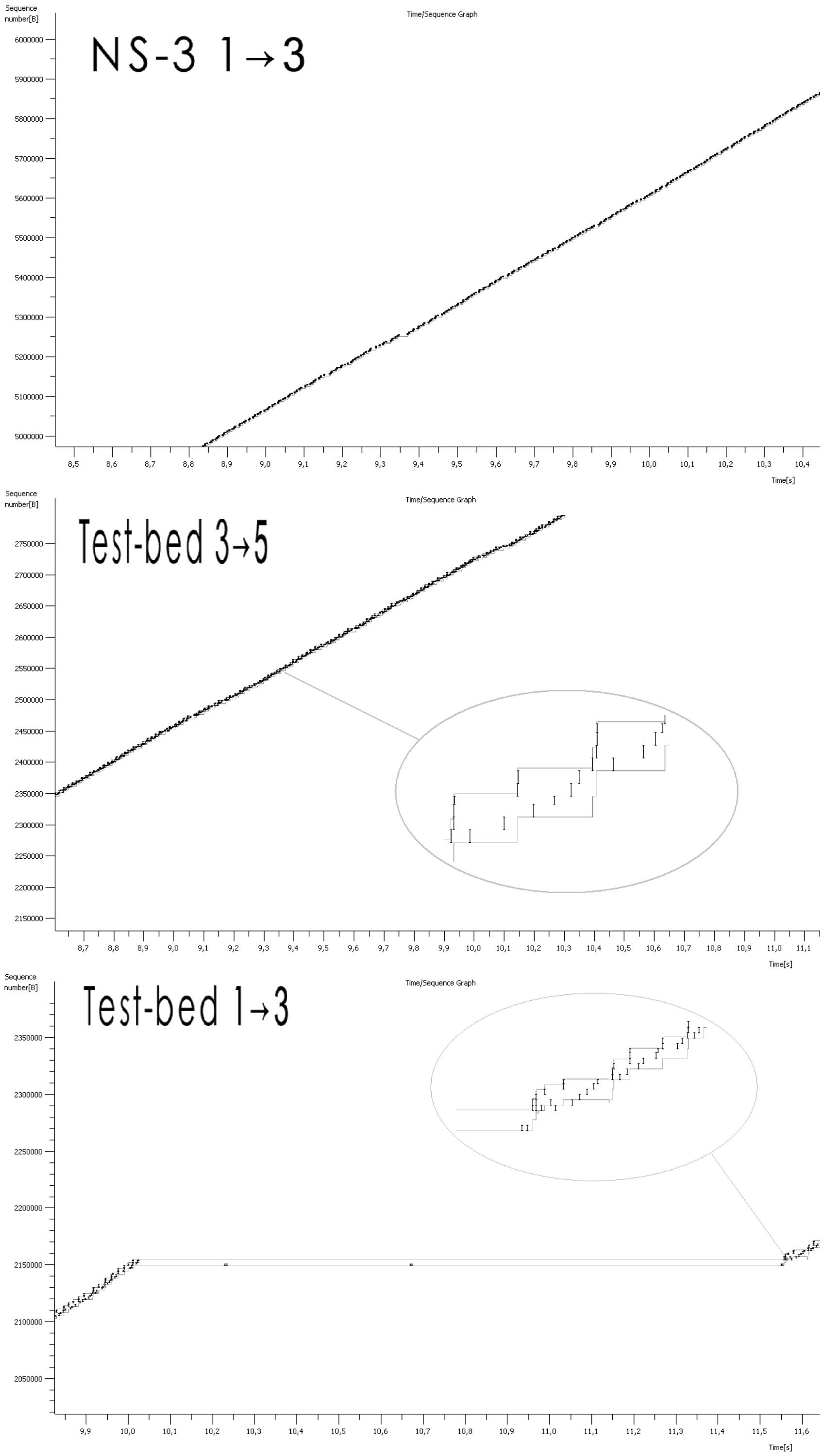

Figure 4.4: RTT from 3-samples PCAP *files*

data in that segment. From the two times, TCP computes RTT. Whenever it obtains a new round trip sample, TCP adjusts its notion of the average round trip time for the connection.

In *Figure 4.4* we can see how in *NS-3* and test-bed 3→5 the behaviour is normal because the sequence number increases linearly with time, but if we look at the test-bed 1→3 graph we can see a big delay and a 3-times retransmission packet. It is shown how the "time out" increases the double each time (there are three black points, between 10 and 11.5 approximately, showing the packet retransmission).

If we pay attention the bubble inside test-bed 3→5 of the graphic, there are two lines when sending packets. The upper line mean "TCP window size" and the downer line mean "Segment Transmitted". We can see how when sending traffic between 3→5 the difference between both lines is big enough, which means that the window size is working fine and there is no any traffic congestion, but if we look at test-bed 1→3, we can see how the difference between lines is reduced. This is because congestion due to interference makes TCP decrease the window size exponentially to avoid this transmission problems, and when traffic begins to flow again, TCP use slow-start (we can see in the bubble inside test-bed 1→3 graph) avoiding swamping the network with additional traffic immediately after the congestion clears.

This congestion is produced in our network by interferences and noises of the environment which produces the decrease of the throughput shown in *Figure 2.21* and in *Figure 3.11*, and the difference between them is due to the fact that there are more interferences in the real world (test-bed) than in the simulation.

Chapter 5

Conclusions

This work has been guided by the vision of creating a WMN test-bed working with the IEEE 802.11s protocol, analyzing and verifying that it was working correctly and explaining the behaviour of the network in the different experiments.

In the first part of this thesis, when we created the test-bed we had some problems installing the Open80211s package because we did not know for sure which tools we needed and also because we were trying to install a package under continuous development and it was complicated to find a stable release. Once we got all the test-bed working, the first experiments were done under some special conditions that allowed us to improve our future experiments.

From the test-bed experiments, we can say that the network is working satisfactorily according to the results obtained. The only point in which the performance is not meeting the expected results is the fact that the behaviour of the network when we were sending traffic from the first node to the last two was not the expected one, because the throughput of the network increased. The reason of these increases was that when we were having traffic problems, we started moving the computers antennas. This made the communication between computers flow better when the traffic problems disappeared.

In the second part of this thesis, implementation in *NS-3* of the test-bed network, we found difficulties implementing interferences and noises. This happened first of all because when we got a WMN working and tried to add a Loss Propagation Model we found that it did not work. After sending some e-mails to the developers and not receiving any answer, we decided to implement it using other methods, always according to the same parameters as the test-bed. When we were finishing this report, we received an answer from the developers, and they helped us fix our implementation problems, but it was too late to test it, we fixed the program and we added it to this report (*Appendix C*) to test it as future work.

Also, when we set the propagation loss model, we could not set all the environment conditions because the *NS-3* models did not allow us. Due to this problem, the *NS-3* implementation does not have exactly the test-bed performance, but we can conclude, according to the results obtained from our experiments, that the test-bed corresponds with *NS-3* implementation behaviour.

As a result, in the one hand we have developed a test-bed where testing different topologies is possible, and on the other hand we have a simulation program for the test-bed which, with only some changes, makes it possible to simulate other topologies or other experiments.

CHAPTER 6

Future work

WMN will be commonly used in a near future, that is why it should be tested and improved before people use it.

In future experiments it would be also interesting to analyze other network details, such as lost packets, round trip delay time, etc. In order to improve this work I have some suggestions for each goal of the thesis.

6.1 Test-bed

First of all, it would be very useful if computers used in the test-bed could be updated with more RAM memory and new USB 2.0 ports to be able to test higher data rates up to 11 Mbps, today's limitation. Also it would be interesting to install the test-bed in a place where lets to test it without having to move all computers around the building, and measuring the interferences that could affect the network in order to control environment variables as much as possible.

Open80211s should be updated to the latest version, because they are improving the system and adding new features that can be tested perfectly in the test-bed. There are also other ideas to carry out:

- Develop a system that allows controlling the entire network from one computer.
- Set the computers with automatic WMN start when anyone turns them on.
- Connect computers to the Internet while they are connected in the mesh, since it is very useful to upgrade the systems continuously.
- Configure the test-bed working with different interfaces (more antennas per node) in different wireless mesh networks.

- Improve the actual test-bed user's manual, adding all the new features.

6.2 *NS-3*

NS-3 is a really useful tool to simulate networks, but it is also possible to combine it with real networks creating huge networks working in real and simulated time. In combination with the test-bed we can also use *NS-3* as a traffic generator, to control and to generate the same traffic for simulation and for the test-bed.

One very important thing to carry out in the future is to develop in *NS-3* a propagation loss model according to different environment conditions. Specifically, develop a propagation loss model using different RSS for each antenna.

Finally, for new simulations, one important task is to use the new *NS-3* IEEE 802.11s draft, and work together with *NS-3* developers to improve the simulator and to make it possible to introduce propagation loss models working correctly (as we have said above when we were finishing this report, we fix meshNet.cc and we attached it to be tested as a future work (see *Appendix C*). Also, another task is to help them to expand the *NS-3* documentation adding new comments that help new users to understand and use the simulator.

Appendix A

Upgrade Ubuntu Kernel

If we have installed on operating system 2.6.28 kernel or less, we should update the kernel to 2.6.29. To update it follow the steps enounced bellow, introducing all the commands on the command line.

A.1 Download all the packets necessary

It is necessary to download all the packets necessaries to install the new kernel for the operative system.

```
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.
6.29.4/linux-headers-2.6.29-02062904-generic\_2.6.
29-02062904\_i386.deb

$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.
6.29.4/linux-headers-2.6.29-02062904\_2.6.29-02062904\
_all.deb

$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.
6.29.4/linux-image-2.6.29-02062904-generic\_2.6.
29-02062904\_i386.deb
```

A.2 Install all the packets

Once all packets are downloaded, install them.

```
$ sudo dpkg -i linux-headers-2.6.29-02062904-generic_2.6.29-02062904_i386.deb
linux-headers-2.6.29-02062904_2.6.29-02062904_all.deb
linux-image-2.6.29-02062904-generic_2.6.29-02062904_i386.deb
```

A.3 Reboot the system

```
$ sudo reboot.
```

A.4 See if you have installed 2.6.29 kernel

Make sure that you have upgraded your kernel.

A.5 Reboot the system

```
$ uname -r
```

A.6 Help

If you want to download and install a Linux kernel higher than 2.6.29 visit this web site: `http://kernel.ubuntu.com/~kernel-ppa/mainline/`

Appendix B

Test-Bed User Manual

This intends to be an User Manual for everybody that wants to use the WMN test-bed we have created.

B.1 Introduction

The test-bed has eight computers. Each computer has an ID, wrote on the CPU, which corresponds to the name of user. The name of user corresponds with the Table b.1 where we can see that each computer has been assigned an IP address and a wireless card that comes with its own MAC address.

The Super User password is the same in all computers: 123456789s

Computer ID	IP address	MAC address
Proyecto 1	192.168.2.1	00:26:5a:01:45:cd
Proyecto 2	192.168.2.2	00:24:01:14:42:26
Proyecto 3	192.168.2.3	00:24:01:14:42:28
Proyecto 4	192.168.2.4	00:24:01:9f:04:d2
Proyecto 5	192.168.2.5	00:24:01:9f:04:cb
Proyecto 6	192.168.2.6	00:24:01:9f:04:e1
Proyecto 7	192.168.2.7	00:24:01:9f:04:c7
Proyecto 8	192.168.2.8	00:24:01:9f:04:63

Table B.1: Computers info (ID, IP, MAC address)

B.2 Configure the network

In each computer it is installed Open80211s package, but when we start the computer, the WMN is not working. If we want to run the network, we have to run a terminal (command line) and access to the folder wireless-testing, that it's on each computer at home folder, and run configure.sh as described on README-CONFIGURE.

README-CONFIGURE

```
Wireless Mesh Network configuration Guide
==================================

This is a set up guide to configure a WMN network using open80211s package. Following
this tutorial you could create a new network or just connect to one already created.

How to configure it
==================================

To configure the network, you should know the name of your wireless interface, the name
of the network you want to connect or create and the IP address you want to assign to
your computer, but remember that it have to be different from any other of the network.

Once you know this information you just have to be on wireless-testing folder and write
the next command:

$ ./configure.sh <interface> <networkName> <computerIp>

For example,

$ ./configure.sh wlan0 mesh 192.168.2.1

After doing this, if everything is right you will configure the network with the parameters
wanted. If you want to configure additional options you just have to add -s to the
command,

$ ./configure.sh <interface> <networkName> <computerIp> -s

and it will ask you to introduce the next parameters,
```

Channel (default on channel 0)
Rate (default 1 Mbps)

Some useful commands
==================================

Manuals useful to set some parameters of the network

$ iw help
$ man ifconfig
$ man iwconfig

Some useful commands to know characteristics of the WMN and how to change some parameters (extracted from above manuals).

-> To know details about the interface or about the parameters setting up of the new interface.

$ ifconfig

-> To know details about your interface card

$ iwconfig <interface>

-> Once you have your network running, you can list the visible nodes from your node and details about the connection.

$ iw dev <networkName> station dump

-> To change the rate

$ sudo iwconfig <networkName> rate <rateValue>

-> To change the channel

$ sudo iw dev <networkName> set channel <channelNum>

Author of this README-CONFIGURE file
==================================

This document has been created by Luis Javier Sánchez Cuenca for his IEEE 802.11s testbed. I hope everybody who wants to configure a WMN can use it.

Configure.sh

```
#!/bin/bash

clear

if [ ! -n ”$3” ]; then

printf ”COMMAND FAILED: Correct use -> $0 interface networkName ipAddress [-s]\n \nread -n 1 option”
printf ”Help: This are the interfaces available: \n \n”
ifconfig -s
exit

fi

printf ”\n\n===*****************************************************************====\n\n\n”
printf ” Configuring Wireless Mesh Network in this computer \n\n\n”
printf ”===*****************************************************************====\n\n”

sudo killall NetworkManager

sudo iw dev $1 interface add $2 type mp mesh_id myMesh

if [ $? -eq 0 ]; then # looks if the command above works right

echo
printf ”Connected to ”
printf $2
echo
echo

else

echo
printf ”Problem connecting to ”
printf $2
printf ”\n \nMaybe it’s already connected, if not try again! \n”

fi

printf
”\n............................................................................\n”
printf ” Info if it’s has been connected correctly \n”
printf ”............................................................................\n\n”

ifconfig -a — grep $2

printf ”\n............................................................................\n”
printf ” Upping the network \n”
printf ”............................................................................\n”

sudo ifconfig $2 $3 up

if [ $? -eq 0 ]; then # looks if the command above works right

echo
printf ”Network ”
printf $2
printf ” up \n”

else
```

```
echo
printf "Problem upping the network "
printf $2
echo
exit
fi

if [ "$4" == "-s" ]; then

printf "\nEnter some parameters \n \n"

printf "\nEnter Channel: "

read -e nchannel

sudo iw dev $2 set channel $nchannel

if [ $? -eq 0 ]; then # looks if the command above works right

echo echo
echo 'Channel configured correctly '
echo
else
echo echo
echo 'Problem setting channel '
echo
exit

fi

printf "\nEnter rate (Ex: 54M): "

read -e nrate

sudo iwconfig $2 rate $nrate

if [ $? -eq 0 ]; then # looks if the command above works right

echo echo
echo 'Rate configured correctly '
echo
else
echo echo
echo 'Problem setting rate '
echo
exit

fi
fi

printf "\n..........................................................................................\n"
printf " Networks info \n"
printf "..........................................................................................\n"

sudo ifconfig
```

B.3 Add a new computer to the test-bed

If we want to add new computers to the test-bed working with the same system, we recommend you to follow the instructions written bellow (in README-INSTALL), but first of all you need to have installed Ubuntu in your computer.

On the CD, there are all the files needed to install the same Open80211s package installed in all the eight computers. There is a folder called "wireless-testing" that contains all the files needed and the ones we have developed to install the package and to configure the network.

README-INSTALL

```
Open80211s Installing Guide
==================================

This is an installation guide to install open80211s package on Linux Operative System.
If you follow the instructions you will have installed this system ready to use it.

What are we going to install?
==================================

We are going to install open80211s package and all packages, programs and libraries
needed.
For this there are 4 steps:

1) Install packages needed to configure kernel options and to choose the driver we need
for our wifi card.
2) Set kernel options
3) Compile the new kernel and install it
4) Install iw and libraries needed to use it.

How to Install it
==================================
First of all you should have a kernel version 2.6.29 or higher. To know what kernel version
you have

$ uname -r
```

This will show you your kernel version if it lower than 2.6.29 update it following the Update kernel section on this document.

Once we have the correct kernel version we have open a new console terminal and write on the command line:

$ cd wireless-testing

$ ./install.sh

After this, on the terminal you will see a menu like this:

-> Choose the option you want

1) Install OPEN80211s package and everything needed
2) Choose the kernel options and right driver
3) Build the kernel and install packages
4) Install iw

HELP: If you choose the first option all will be installed and if you choose other option, options forward will be executed also"

****
Option:

If you choose the first one all the system will be installed in your computer if you follow the steps. If you have any problem, this program will show you the problem and you can start installing again from the last step you were, choosing between 2 and 4 options depending the installing phase you were.

To choose kernel options you will have to know which driver it's the one of your driver card. With this command you can know which the driver of your card is,

$ sudo lshw -C Network

If you look at "Driver settings" on this document you will see the kernel options you have to choose on kernel menu for your driver (there are only a few of them, if you don't find yours just look on `http://www.open80211s.org/` and look for your wireless card driver) .

At the end of this program you will have installed Open80211s package and everything you need to create a WMN.

To create a WMN I recommend you to read /wireless-testing/README-CONFIGURE

Update kernel
==================================

To update the kernel you have to download all Linux packets necessaries, the next steps it's to update up 2.6.29 but if you want to upgrade it to another version just visit `http://kernel.ubuntu.com/~kernel-ppa/mainline/` and download the version you want.

$ wget `http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.6.29.4/linux-headers-2.6.29-02062904-generic\_2.6.29-02062904\_i386.deb`
$ wget `http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.6.29.4/linux-headers-2.6.29-02062904\_2.6.29-02062904\_all.deb`
$ wget `http://kernel.ubuntu.com/~kernel-ppa/mainline/v2.6.29.4/linux-image-2.6.29-02062904-generic\_2.6.29-02062904\_i386.deb`

After download the packages install them using

$ sudo dpkg -i linux-headers-2.6.29-02062904-generic_2.6.29-02062904_i386.deb linux-headers-2.6.29-02062904_2.6.29-02062904_all.deb
linux-image-2.6.29-02062904-generic_2.6.29-02062904_i386.deb

Once you have installed it reboot your system and check that you have installed it correctly (check that its the correct kernel version)

$ sudo reboot.

Driver settings
==================================

On this section i going to show some drivers (as a example) working with open80211s and the options you have to choose on kernel menu. If you don't find your driver I recommend you to look on `http://www.open80211s.org/`
For all the drivers you have to enable mac80211:
——————-
Networking —>
Wireless —>
<M> Improved wireless configuration API
<M> Generic IEEE 802.11 Networking Stack (mac80211)

ath5k
******
Enable ath5k in the kernel
———————-
Device Drivers —> Network device support —> Wireless LAN —>
<M> Atheros 5xxx wireless cards support

ath9k
******
Enable ath9k in the kernel
———————-
Device Drivers —> Network device support —> Wireless LAN —>
<M> Atheros 802.11n wireless cards support

libertas_tf
******
Enable libertas_tf in the kernel
———————-
Networking —> Wireless —>
<M> Marvell 8xxx Libertas WLAN driver support with thin firmware
<M> Marvell Libertas 8388 USB 802.11b/g cards with thin firmware

p54
******
Enable p54 in the kernel
———————-
Device Drivers —> Network device support —> Wireless LAN —> Wireless LAN (IEEE 802.11)
<M> Softmac Prism54 support
<M> Prism54 USB support
<M> Prism54 PCI support
<M> Prism54 SPI (stlc45xx) support

Author of this README-INSTALL file
=================================
This document has been created by Luis Javier Sánchez Cuenca for his IEEE 802.11s testbed. I hope everybody who wants to create a WMN can use it.

Install.sh

```
#!/bin/bash
clear

printf ”\n\n===*****************************************************************====\n\n\n”
printf ” Installig OPEN80211S \n\n\n”
printf ”===*****************************************************************====\n\n\n”
printf ”-¿ Choose the option you want \n\n”
printf ”1) Install OPEN80211s package and everything needed \n”
printf ”2) Choose the kernel options and right driver \n”
printf ”3) Build the kernel and install packages\n”
printf ”4) Install iw \n\n”
printf ”HELP: If you choose the first option all will be installed and if you choose\n”
printf ” other option, options forward will be executed also \n\n”
printf ”********\n”
printf ”Option: ”
read -n 1 option

if [ $option -gt 4 ] ; then
echo
echo ”\n Option should be between 1 and 4, bye.”
exit 1
fi
#————————————————————————————————
if [ $option -eq 1 ] ; then
printf ”\n ..........................................................................................\n”
printf ” Installing the following packages: fakeroot build-essential git-core kernel-package \n”
printf ”..........................................................................................\n”

# fakeroot
sudo sudo apt-get install -y fakeroot

if [ $? -eq 0 ]; then # looks if the command above works right
echo
echo ’Fakeroot installed correctly’
echo
else
echo
echo ’Problem installing fakeroot’
exit
fi

# build-essential
sudo sudo apt-get install -y build-essential


if [ $? -eq 0 ]; then # looks if the command above works right
echo
echo ’Build-essential installed correctly’
echo
else
echo
echo ’Problem installing Build-essential’
exit
fi
```

```
# git-core
sudo sudo apt-get install -y git-core

if [ $? -eq 0 ]; then # looks if the command above works right
echo
echo 'git-core installed correctly'
echo
else
echo
echo 'Problem installing git-core'
exit
fi

# kernel-package
sudo sudo apt-get install -y kernel-package

if [ $? -eq 0 ]; then # looks if the command above works right
echo
echo 'Kernel-package installed correctly'
echo
else
echo
echo 'Problem installing kernel-package'
exit
fi
printf " Installing ncourses \n\n"

# libncurses5
sudo apt-get install libncurses5

if [ $? -eq 0 ]; then # looks if the command above works right
echo
echo 'Libncurses5 installed correctly'
echo
else
echo
echo 'Problem installing libncurses5'
exit
fi


# ncurses-dev

sudo apt-get install ncurses-dev

if [ $? -eq 0 ]; then # looks if the command above works right

echo echo 'ncurses-dev installed correctly'

echo

else

echo echo 'Problem installing ncurses-dev'

exit

fi

echo

echo

fi

#————————————————————————————————
```

```
if [ $option -lt 3 ] ; then
printf "\n ..........................................................................................\n"
printf " Copying current kernel's configuration \n"
printf "..........................................................................................\n"
sudo cp /boot/config-`uname -r` .config
sudo make menuconfig
fi

#————————————————————————————————————————

if [ $option -lt 4 ] ; then
printf "\n ..........................................................................................\n"
printf " Building the kernel and making packages \n"
printf "..........................................................................................\n"

# sudo fakeroot make-kpkg –initrd kernel_image kernel_headers

if [ $? -eq 0 ]; then # looks if the command above works right
echo echo 'Kernel built and packages made'
echo
else
echo echo 'Problem building the kernel and making packages'
echo
exit
fi
printf "..........................................................................................\n"
printf " Installing packages \n"
printf "..........................................................................................\n"
cd ..

# sudo dpkg -i linux-*.deb

if [ $? -eq 0 ]; then # looks if the command above works right
echo echo 'Packages installed'
echo
else
echo echo 'Problem installing packages'
echo
exit
fi
fi
#————————————————————————————————————————
if [ $option -lt 5 ] ; then
cd iw-0.9.17
printf "\n ..........................................................................................\n"
printf " Updating libraries for iw \n"
printf "..........................................................................................\n"
sudo apt-get install libnl1 libnl-doc libnl-dev


if [ $? -eq 0 ]; then # looks if the command above works right

echo echo 'Libraries updated'

echo else

echo echo 'Problem updating libraries: libnl1 libnl-doc libnl-dev '

echo exit

fi
```

```
printf "..................................................................................\n"
printf " Installing iw \n"
printf "..................................................................................\n"

sudo make install

if [ $? -eq 0 ]; then # looks if the command above works right

echo
echo 'iw installed'
echo

else

echo
echo 'Problem installing iw '
echo
exit

fi

fi

#--------------------------------------------------------------------------
printf "\n\n===*****************************************************************====\n\n"
printf " Congratulations!! OPEN80211S has been installed correctly!! \n\n\n"
printf "===*****************************************************************====\n\n\n"
```

Appendix C

NS-3 Program

In this appendix it is shown the program developed to simulate the test-bed network on NS-3, an example of script used to run this program and also the NS-3 program implemented using IEEE 802.11s protocol. The program works with the class MeshTest (Figure c.1) that offers us the possibility to create, configure and run our network.

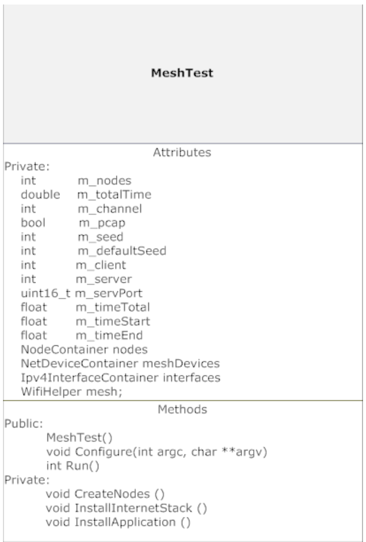

Figure C.1: MestTest class

To run this program on NS-3 simulator we have to know all the input parameters we can set, here is the list of parameters we can set:

- **nodes**: Choose the number of nodes in the network [Default 8]
- **server**: Choose which node the server is [Default 1]

- **client**: Choose which node the client is [Default 0]
- **time**: Choose the simulation time in seconds [120 s]
- **channel**: Select channel of transmission [Default 3]
- **pcap**: Enable PCAP traces on interfaces [Default 0]
- **seed**: Set the seed for random values [Default 9]

If we introduce in terminal this command ./waf –run "b11mesh –PrintHelp", it will display all this parameters. An example to run this program with some of these parameters is:

./waf -run "b11mesh –pcap=1 –client=0 –server=6 –channel=6 –time=60"

This command will run the program that will simulate the network working on channel 6 and sending traffic from node 0 to node 6 during 60 seconds. When simulation finish, it will give as a result one ".pcap" file per node, because the pcap option is activated. Run this program without any parameters (./waf –run "b11mesh") or without defining all parameters, the program will run using defaults parameters.

C.1 *b11mesh.cc*

This is the code of the program that simulates using *NS-3* the test-bed using the protocol IEEE 802.11b. This program has been used to make the *NS-3* experiments.

```
/* -*- Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
* ./waf –run "b11mesh –pcap=1 –seed=23 –client=0 –server=1"
*
* 37.96m 36.65m 36.65m 33.79m 39.21m 25.41m 44.10m
* * ___ * ___ * ___ * ___ * ___ * ___ * ___ *
* node1 node2 node3 node4 node5 node6 node7 node8
*/

#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
#include "ns3/global-routing-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mesh-module.h"
#include "ns3/mobility-module.h"
#include "ns3/mesh-helper.h"
#include <iostream>
#include <sstream>
#include <fstream>


using namespace ns3;
```

```
NS_LOG_COMPONENT_DEFINE ("TestMeshScript");

class MeshTest
{
public:
/// Init test
MeshTest ();
/// Configure test from command line arguments
void Configure (int argc, char ** argv);
/// Run test
int Run ();
private:

int m_nodes;
double m_totalTime;
int m_channel;
bool m_pcap;
int m_seed; // seed for random values
int m_defaultSeed;
// Client and server

int m_client;
int m_server;
uint16_t m_servPort;

// Calculate time of simulation
float m_timeTotal,m_timeStart,m_timeEnd;

/// List of network nodes
NodeContainer nodes;
/// List of all mesh point devices
NetDeviceContainer meshDevices;
//Addresses of interfaces:
Ipv4InterfaceContainer interfaces;
// MeshHelper. Report is not static methods
WifiHelper mesh;

private:
/// Create nodes and setup their mobility
void CreateNodes ();
/// Install internet m_stack on nodes
void InstallInternetStack ();
/// Install applications
void InstallApplication ();


};
MeshTest::MeshTest () :
m_nodes (8), m_totalTime (120.0),
m_channel (3), m_pcap (false),
m_seed (23), m_defaultSeed (9),
m_client (0), m_server (1),
m_servPort (5001) {
}
```

```
void
MeshTest::Configure (int argc, char *argv[ ])
{
CommandLine cmd;

// enable rts cts all the time.

SeedManager::SetSeed(m_defaultSeed); // Change the default value of the seed

Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",StringValue ("3000"));
// disable fragmentation
Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("3000"));

cmd.AddValue ("nodes", "Number of nodes in the network [Default 8]", m_nodes);
cmd.AddValue ("server", "Node server [Default 1]", m_server);
cmd.AddValue ("client", "Node client [Default 0]", m_client);
cmd.AddValue ("time", "Simulation time, seconds [120 s]", m_totalTime);
cmd.AddValue ("channel", "Select channel of transmision [Default 3]", m_channel);
cmd.AddValue ("pcap", "Enable PCAP traces on interfaces. [Default 1]", m_pcap);
cmd.AddValue ("seed", "Set the seed for random values [Default 9]", m_seed);

cmd.Parse (argc, argv);
NS_LOG_DEBUG ("Number of nodes: " << m_nodes);
NS_LOG_DEBUG ("Simulation time: " << m_totalTime << " s");
}

void
MeshTest::CreateNodes ()
{
// Parameters Loss Propagation Model

double m_referenceDistance = 1.0; // m
double m_exponent = 1.7;
double m_referenceLoss = 40.178882771; // L0 = 20 log(4 pi feqChann3 / 3 exp8)
double m_EnergyDet = -86.0;
double m_ccath = -99.0;
double m_txpower = 18.0; // dbm

SeedManager::SetRun(m_seed);

NS_LOG_INFO ("Building chain topology.");

nodes.Create (m_nodes);
mesh.SetStandard (WIFI_PHY_STANDARD_80211b);

NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();

// Configure YansWifiChannel
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();

wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (m_EnergyDet)); //defulat val is -96dBm
wifiPhy.Set ("CcaMode1Threshold", DoubleValue (m_ccath)); //default val is -99dBm
wifiPhy.Set ("TxGain", DoubleValue (1.0)); // Use 5.0 to extend range to about 300 meters
wifiPhy.Set ("RxGain", DoubleValue (1.0)); // Use 5.0 to extend range to about 300 meters
wifiPhy.Set ("TxPowerLevels", UintegerValue (1) );
wifiPhy.Set ("TxPowerEnd", DoubleValue (m_txpower) ); //default val is 16.0206dBm
wifiPhy.Set ("TxPowerStart", DoubleValue (m_txpower) ); //default val is 16.0206dBm
wifiPhy.Set ("RxNoiseFigure", DoubleValue (7.0) ); //defulat val is 7dB
wifiPhy.Set ("ChannelNumber", UintegerValue(m_channel));


wifiPhy.SetPcapFormat (YansWifiPhyHelper::PCAP_FORMAT_80211_RADIOTAP);
```

```
YansWifiChannelHelper wifiChannel;

wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel","Exponent",DoubleValue(m_exponent),
"ReferenceDistance",DoubleValue(m_referenceDistance),"ReferenceLoss",DoubleValue(m_referenceLoss));

wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel","Speed",DoubleValue(3.0e8));

wifiChannel.AddPropagationLoss("ns3::RandomPropagationLossModel","Variable",RandomVariableValue
(UniformVariable(27.0, 43.0) ));

wifiPhy.SetChannel (wifiChannel.Create ());
Ssid ssid = Ssid ("wifi-default");
mesh.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode", StringValue ("wifib-11mbs"));

wifiMac.SetType ("ns3::AdhocWifiMac");

meshDevices = mesh.Install (wifiPhy, wifiMac, nodes);

MobilityHelper mobility;

ns3::ListPositionAllocator *listPosNod = new ns3::ListPositionAllocator();

listPosNod->Add(*new ns3::Vector3D::Vector3D(0.0,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(37.96,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(74.61,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(111.26,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(145.05,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(184.26,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(209.67,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(253.77,0.0,0.0));

mobility.SetPositionAllocator(listPosNod);

mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (nodes);

if (m_pcap)
wifiPhy.EnablePcapAll (std::string ("mp-"));
}
void
MeshTest::InstallInternetStack ()
{
NS_LOG_INFO (" Installing internet stack on all nodes and assigning IP Addresses.");

InternetStackHelper internetStack;
internetStack.Install (nodes);

Ipv4AddressHelper address;
address.SetBase ("192.168.2.0", "255.255.255.0");
interfaces = address.Assign (meshDevices);

Ipv4StaticRoutingHelper ipv4RoutingHelper;
Ptr<Ipv4> ipv4Node;
Ptr<Ipv4StaticRouting> staticRoutingNode;

int nod = 0;

while (nod < m_nodes){

ipv4Node = nodes.Get(nod)->GetObject<Ipv4> ();
staticRoutingNode = ipv4RoutingHelper.GetStaticRouting (ipv4Node);
```

```
int leftNodes = nod - 1;
int leftExit = leftNodes;

while (leftNodes>=0) {

staticRoutingNode->AddHostRouteTo (interfaces.GetAddress (leftNodes), interfaces.GetAddress (leftExit), 1);
leftNodes–;

}

int rightNodes = nod + 1;
int rightExit = rightNodes;

while (rightNodes<=m_nodes-1) {

staticRoutingNode->AddHostRouteTo (interfaces.GetAddress (rightNodes), interfaces.GetAddress (rightExit), 1);
rightNodes++;

}

nod++;
}

}

void
MeshTest::InstallApplication ()
{ NS_LOG_INFO (”Create applications.”);

uint32_t nPackets = 1500;

// enable rts cts all the time.
Config::SetDefault (”ns3::WifiRemoteStationManager::RtsCtsThreshold”,StringValue (”6000”));
// disable fragmentation
Config::SetDefault (”ns3::WifiRemoteStationManager::FragmentationThreshold”, StringValue (”6000”));
Config::SetDefault (”ns3::TcpSocket::SegmentSize”, UintegerValue (1460));
Config::SetDefault (”ns3::TcpSocket::RcvBufSize”, UintegerValue (900000));
Config::SetDefault (”ns3::TcpSocket::SndBufSize”, UintegerValue (900000));

Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny (), m_servPort));
PacketSinkHelper sinkHelper (”ns3::TcpSocketFactory”, sinkLocalAddress);

ApplicationContainer sinkApp = sinkHelper.Install (nodes.Get (m_server));

sinkApp.Start (Seconds (0.001));
sinkApp.Stop (Seconds (m_totalTime));

// Create the OnOff applications to send TCP to the server
OnOffHelper sourceHelper (”ns3::TcpSocketFactory”, Address ());
sourceHelper.SetAttribute (”OnTime”, RandomVariableValue (ConstantVariable (m_totalTime)));
sourceHelper.SetAttribute (”OffTime”, RandomVariableValue (ConstantVariable (0)));

AddressValue remoteAddress (InetSocketAddress (interfaces.GetAddress (m_server), m_servPort));

sourceHelper.SetAttribute (”Remote”, remoteAddress);
sourceHelper.SetAttribute (”DataRate”, DataRateValue(DataRate(”11Mbps”)));
sourceHelper.SetAttribute (”PacketSize”, UintegerValue (nPackets));


ApplicationContainer sourceApp;
```

```
sourceApp.Add (sourceHelper.Install (nodes.Get(m_client)));
sourceApp.Start (Seconds (0.005));
sourceApp.Stop (Seconds (m_totalTime));

}
int
MeshTest::Run ()
{
CreateNodes ();
InstallInternetStack ();
InstallApplication ();

m_timeStart=clock();

Simulator::Stop (Seconds (m_totalTime));
Simulator::Run ();
Simulator::Destroy ();

m_timeEnd=clock();

m_timeTotal=(m_timeEnd - m_timeStart)/(double) CLOCKS_PER_SEC;

std::cout << "The time of the simulation is: " << m_timeTotal << "\n";

return 0;
}


int
main (int argc, char *argv[])
{
MeshTest t;
t.Configure (argc, argv);
return t.Run();
}
```

C.2 *runMeshTest1.cc*

This is an example script to run the simulation test, this one is to run test 1.

```
#!/bin/sh
############################################################
####
#### TEST 1 (1 HOPS)
####
############################################################
#init variables
```

```
TRIALS="1 2 3 4 5 6"
NODES="0 1 2 3 4 5 6"
NEXTNODE=0
SEED=9

echo Run TEST 1

#run experiments
before="$(date +%s)"

for node in $NODES
do
NEXTNODE=$(($node+1))

for trial in $TRIALS
do

SEED=$(($SEED+$node+5))

echo
echo From $node to $NEXTNODE, Trial $trial, with seed $SEED
echo

../waf –run "b11mesh –pcap=1 –client=$node –server=$NEXTNODE –seed=$SEED"
cp ../mp–$node-0.pcap ../../../../experimentsMesh/test1/client–$node-$NEXTNODE–trial$trial.pcap
cp ../mp–$NEXTNODE-0.pcap ../../../../experimentsMesh/test1/server–$node-$NEXTNODE–trial$trial.pcap

done
done

after="$(date +%s)"
elapsed_seconds="$(expr $after - $before)"
echo Elapsed time testing 1: $elapsed_seconds


echo

echo "Done mesh TEST 1"
```

C.3 *meshNet.cc*

This is the code of the program that simulates using *NS-3* the test-bed using the protocol IEEE 802.11s.

```
/* -*- Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
* 37.96m 36.65m 36.65m 33.79m 39.21m 25.41m 44.10m
* * ___ * ___ * ___ * ___ * ___ * ___ * ___ *
* node1 node2 node3 node4 node5 node6 node7 node8
*/
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
```

```
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
#include "ns3/global-routing-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mesh-module.h"
#include "ns3/mobility-module.h"
#include "ns3/mesh-helper.h"
#include <iostream>
#include <sstream>
#include <fstream>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("TestMeshScript");

class MeshTest
{
public:
/// Init test
MeshTest ();
/// Configure test from command line arguments
void Configure (int argc, char ** argv);
/// Run test
int Run ();
private:

int m_nodes;
double m_randomStart;
double m_totalTime;
int m_channel;
bool m_pcap;
std::string m_stack;
std::string m_root;
int m_seed; // seed for random values
// Cliente and server

int m_client;
int m_server;
uint16_t m_servPort;

// Calculate time of simulation
float m_timeTotal,m_timeStart,m_timeEnd;

/// List of network nodes
NodeContainer nodes;
/// List of all mesh point devices
NetDeviceContainer meshDevices;
//Addresses of interfaces:
Ipv4InterfaceContainer interfaces;
// MeshHelper. Report is not static methods
MeshHelper mesh;


private:

/// Create nodes and setup their mobility

void CreateNodes ();

/// Install internet m_stack on nodes

void InstallInternetStack ();

/// Install applications

void InstallApplication ();

};
```

```
MeshTest::MeshTest () :

m_nodes (8),
m_randomStart (0.1),
m_totalTime (120.0),
m_channel (3),
m_pcap (false),
m_stack ("ns3::Dot11sStack"),
m_root ("ff:ff:ff:ff:ff:ff"),
m_seed (9),
m_client (0),
m_server (1),
m_servPort (5001)
{
}
void
MeshTest::Configure (int argc, char *argv[])
{
CommandLine cmd;
/*
* As soon as starting node means that it sends a beacon,
* simultaneous start is not good.
*/

cmd.AddValue ("nodes", "Number of nodes in the network", m_nodes);
cmd.AddValue ("server", "Node server", m_server);
cmd.AddValue ("client", "Node client", m_client);
cmd.AddValue ("start", "Maximum random start delay, seconds. [0.1 s]", m_randomStart);
cmd.AddValue ("time", "Simulation time, seconds [100 s]", m_totalTime);
cmd.AddValue ("channel", "Select channel of transmision", m_channel);
cmd.AddValue ("pcap", "Enable PCAP traces on interfaces. [0]", m_pcap);
cmd.AddValue ("seed", "Set the seed for random values", m_seed);
cmd.AddValue ("stack", "Type of protocol stack. ns3::Dot11sStack by default", m_stack);
cmd.AddValue ("root", "Mac address of root mesh point in HWMP", m_root);

cmd.Parse (argc, argv);
NS_LOG_DEBUG ("Number of nodes: " << m_nodes);
NS_LOG_DEBUG ("Simulation time: " << m_totalTime << " s");
}

void
MeshTest::CreateNodes ()
{
// Parameters Loss Propagation Model

double m_referenceDistance = 1.0; // m
double m_exponent = 1.7;
double m_referenceLoss = 40.178882771; // L0 = 20 log(4 pi feqChann3 / 3 exp8)
double m_EnergyDet = -88.0;
double m_ccath = -99.0;
double m_txpower = 18.0; // dbm

NS_LOG_INFO ("Building chain topology.");

SeedManager::SetRun(m_seed);

nodes.Create (m_nodes);
```

```
// Configure YansWifiChannel
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();

wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (m_EnergyDet)); //defulat val is -96dBm
wifiPhy.Set ("CcaMode1Threshold", DoubleValue (m_ccath)); //default val is -99dBm
wifiPhy.Set ("TxGain", DoubleValue (1.0)); // Use 5.0 to extend range to about 300 meters
wifiPhy.Set ("RxGain", DoubleValue (1.0)); // Use 5.0 to extend range to about 300 meters
wifiPhy.Set ("TxPowerLevels", UintegerValue (1) );
wifiPhy.Set ("TxPowerEnd", DoubleValue (m_txpower) ); //default val is 16.0206dBm
wifiPhy.Set ("TxPowerStart", DoubleValue (m_txpower) ); //default val is 16.0206dBm
wifiPhy.Set ("RxNoiseFigure", DoubleValue (7.0) ); //defulat val is 7dB
wifiPhy.Set ("ChannelNumber", UintegerValue(m_channel));

YansWifiChannelHelper wifiChannel;

wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel","Exponent",DoubleValue(m_exponent),
"ReferenceDistance",DoubleValue(m_referenceDistance),"ReferenceLoss",DoubleValue(m_referenceLoss));

wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel","Speed",DoubleValue(3.0e8));

wifiChannel.AddPropagationLoss("ns3::RandomPropagationLossModel","Variable",RandomVariableValue
(UniformVariable(27.0, 43.0) ));

wifiPhy.SetChannel (wifiChannel.Create ());
/*
* Create mesh helper and set stack installer to it
* Stack installer creates all needed protocols and install them to
* mesh point device
*/
mesh = MeshHelper::Default ();

mesh.SetStackInstaller (m_stack);

mesh.SetMacType ("RandomStart", TimeValue (Seconds(m_randomStart)));
// Set number of interfaces - default is single-interface mesh point
mesh.SetNumberOfInterfaces (1);
// Install protocols and return container if MeshPointDevices
meshDevices = mesh.Install (wifiPhy, nodes);
// Setup mobility - static chain topology

MobilityHelper mobility;

ns3::ListPositionAllocator *listPosNod = new ns3::ListPositionAllocator();

listPosNod->Add(*new ns3::Vector3D::Vector3D(0.0,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(37.96,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(74.61,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(111.26,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(145.05,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(184.26,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(209.67,0.0,0.0));
listPosNod->Add(*new ns3::Vector3D::Vector3D(253.77,0.0,0.0));

mobility.SetPositionAllocator(listPosNod);

mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (nodes);


if (m_pcap)

wifiPhy.EnablePcapAll (std::string ("mp-"));

}
```

```
void
MeshTest::InstallInternetStack ()
{
NS_LOG_INFO (" Installing internet stack on all nodes and assigning IP Addresses.");

InternetStackHelper internetStack;
internetStack.Install (nodes);

Ipv4AddressHelper address;
address.SetBase ("192.168.2.0", "255.255.255.0");
interfaces = address.Assign (meshDevices);

Ipv4StaticRoutingHelper ipv4RoutingHelper;
Ptr<Ipv4> ipv4Node;
Ptr<Ipv4StaticRouting> staticRoutingNode;

int nod = 0;

while (nod < m_nodes){

ipv4Node = nodes.Get(nod)->GetObject<Ipv4> ();
staticRoutingNode = ipv4RoutingHelper.GetStaticRouting (ipv4Node);

int leftNodes = nod - 1;
int leftExit = leftNodes;

while (leftNodes>=0) {

staticRoutingNode->AddHostRouteTo (interfaces.GetAddress (leftNodes), interfaces.GetAddress (leftExit), 1);
leftNodes–;

}

int rightNodes = nod + 1;
int rightExit = rightNodes;

while (rightNodes<=m_nodes-1) {

staticRoutingNode->AddHostRouteTo (interfaces.GetAddress (rightNodes), interfaces.GetAddress (rightExit), 1);
rightNodes++;

}

nod++;
}

}

void
MeshTest::InstallApplication ()
{ NS_LOG_INFO ("Create applications.");

uint32_t nPackets = 1500;

// enable rts cts all the time.
Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",StringValue ("6000"));


// disable fragmentation

Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("6000"));
```

```
Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue (1460));
Config::SetDefault ("ns3::TcpSocket::RcvBufSize", UintegerValue (900000));
Config::SetDefault ("ns3::TcpSocket::SndBufSize", UintegerValue (900000));

Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny (), m_servPort));
PacketSinkHelper sinkHelper ("ns3::TcpSocketFactory", sinkLocalAddress);

ApplicationContainer sinkApp = sinkHelper.Install (nodes.Get (m_server));

sinkApp.Start (Seconds (0.001));
sinkApp.Stop (Seconds (m_totalTime));

// Create the OnOff applications to send TCP to the server
OnOffHelper sourceHelper ("ns3::TcpSocketFactory", Address ());
sourceHelper.SetAttribute ("OnTime", RandomVariableValue (ConstantVariable (m_totalTime)));
sourceHelper.SetAttribute ("OffTime", RandomVariableValue (ConstantVariable (0)));

AddressValue remoteAddress (InetSocketAddress (interfaces.GetAddress (m_server), m_servPort));

sourceHelper.SetAttribute ("Remote", remoteAddress);
sourceHelper.SetAttribute ("DataRate", DataRateValue(DataRate("11Mbps")));
sourceHelper.SetAttribute ("PacketSize", UintegerValue (nPackets));

ApplicationContainer sourceApp;

sourceApp.Add (sourceHelper.Install (nodes.Get(m_client)));
sourceApp.Start (Seconds (0.005));
sourceApp.Stop (Seconds (m_totalTime));

}
int
MeshTest::Run ()
{
CreateNodes ();
InstallInternetStack ();
InstallApplication ();

m_timeStart=clock();
Simulator::Stop (Seconds (m_totalTime));
Simulator::Run ();
Simulator::Destroy ();

m_timeEnd=clock();
m_timeTotal=(m_timeEnd - m_timeStart)/(double) CLOCKS_PER_SEC;

std::cout << "The time of the simulation is: " << m_timeTotal << "\n";

return 0;
}

int main (int argc, char *argv[])

{

MeshTest t;

t.Configure (argc, argv);

return t.Run();

}
```

APPENDIX D

Throughput tables

In this appendix we shows the tables with the throughput values obtained in the experiment I, in the experiment II and in the *NS-3* simulations.

D.1 Test-bed experiment I

		UDP							TCP						
		Number of Hops													
		1	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**2**	**3**	**4**	**5**	6	**7**
Nº of Trial	**1**	45806	29525	45407	35356	35527	31881	27068	602	793	502	247	468	236	214
	2	3130	41005	34135	45365	45566	45517	32284	513	954	317	356	249	281	358
	3	29326	54170	15662	29589	37557	36572	11020	1095	777	34.6	285	223	180	327
	4	54193	17308	39570	45179	45452	36502	25705	1446	908	424	243	194	153	209
	5	28810	28247	33565	45867	45051	17271	46084	1384	851	597	362	160	278	279
	Average	**32253**	**34051**	**33667.8**	**40271.2**	**41830.6**	**33548.6**	**28432.2**	**1008**	**856.6**	**374.92**	**298.6**	**258.8**	**225.6**	**277.4**

Table D.1: UDP and TCP throughput (bps) of the network

D.2 Test-bed experiment II

		1 HOP						
		1 → 2	**2 → 3**	**3 → 4**	**4 → 5**	**5 → 6**	**6 → 7**	**7 → 8**
Nº of Trial	**1**	4.4585246	4.842981	4.3448787	4.8888169	4.5135569	4.9321181	4.955809
	2	4.7021687	4.9382767	4.7012998	4.9651791	4.4297799	4.8496237	4.8208104
	3	4.7986196	4.6499251	4.8295468	4.9368653	4.5329265	4.8171865	4.9241896
	4	4.7826222	4.6113011	4.82504	4.9575389	4.6505878	4.8251462	4.9729845
	5	4.7783884	4.5951114	4.8241744	4.9723945	4.3479207	4.8239992	4.971514
	6	4.6334699	4.6503149	4.8385495	4.9718446	4.751206	4.9708296	4.9795527
	Average						**4.775418413**	

Table D.2: Throughput test-bed experiment II 1 hop

		2 HOPS					
		1 → 3	**2 → 4**	**3 → 5**	**4 → 6**	**5 → 7**	**6 → 8**
Nº of Trial	**1**	3.20131	3.3439365	4.566476	3.9976141	3.4019898	3.7959282
	2	3.2793217	3.2968881	4.569609	3.8697755	3.5309798	4.0753385
	3	3.1289598	3.468253	4.3549925	3.9693147	3.3747914	3.7628697
	4	3.0060787	3.4388449	4.5712135	3.8401572	3.4853638	4.1810284
	5	3.1423997	3.4133039	4.5503306	3.8969719	3.5857853	4.2024686
	6	2.9210881	3.470862	4.5734008	3.9671889	3.4738473	4.1858265
	Average					**3.747069677**	

Table D.3: Throughput test-bed experiment II 2 hops

		3 HOPS				
		1 → 4	**2 → 5**	**3 → 6**	**4 → 7**	**5 → 8**
Nº of Trial	**1**	2.3244106	2.7162679	2.3240352	2.225909	2.4924817
	2	2.0166297	2.8344546	2.3381739	2.2414781	2.2208648
	3	2.1201052	2.739242	2.3939475	2.0997728	2.2454141
	4	2.130249	2.6863344	2.5122025	1.9673281	2.044254
	5	2.1627872	2.9236249	2.4570673	2.188172	2.5261997
	6	2.3434461	2.7436328	2.3888715	2.1802041	2.3059017
	Average				**2.363115412**	

Table D.4: Throughput test-bed experiment II 3 hops

		4 HOPS			
		1 → 5	**2 → 6**	**3 → 7**	**4 → 8**
Nº of Trial	**1**	2.0950782	2.3874142	1.8449672	1.1918849
	2	2.2259702	2.0272332	1.9426842	2.221281
	3	1.4905348	2.3486063	1.9135667	2.2634781
	4	2.2303381	2.2275728	1.8535718	2.2421399
	5	1.545044	1.9063498	1.9067834	2.1770795
	6	1.3105291	2.0561436	1.8587855	2.0257312
	Average			**1.970531983**	

Table D.5: Throughput test-bed experiment II 4 hops

		5 HOPS		
		1 → 6	**2 → 7**	**3 → 8**
Nº of Trial	**1**	1.3577809	1.9071112	1.8601364
	2	1.3266903	0.3382282	1.9622732
	3	1.4378166	1.7672082	1.5099481
	4	1.2812829	0.4568496	1.9515229
	5	1.3379841	0.5795886	1.4626714
	6	1.2321649	1.8145077	1.933446
	Average		**1.417622845**	

Table D.6: Throughput test-bed experiment II 5 hops

		6 HOPS	
		1 → 7	**2 → 8**
Nº of Trial	**1**	0.4802112	1.8884402
	2	1.6589956	1.8066262
	3	0.3990097	1.4002415
	4	1.6030446	1.8964828
	5	1.6898207	1.7687247
	6	1.6616146	1.8173666
	Average		**1.5058815**

Table D.7: Throughput test-bed experiment II 6 hops

		7 HOPS
		1 → 8
Nº of Trial	**1**	1.3784905
	2	1.6489115
	3	1.4235967
	4	1.6167096
	5	1.7320821
	6	1.6721199
	Average	**1.5786517**

Table D.8: Throughput test-bed experiment II 7 hops

D.3 NS-3 experiment

		1 HOP						
		1 → 2	**2 → 3**	**3 → 4**	**4 → 5**	**5 → 6**	**6 → 7**	**7 → 8**
Nº of Trial	**1**	4.6464768	4.7909452	4.7448575	5.0260253	4.4755622	5.6489047	4.1397154
	2	4.6001521	4.7419624	4.7695319	5.0250635	4.5284812	5.6345024	4.065551
	3	4.6088556	4.753449	4.6771403	4.9864103	4.4721973	5.6465083	4.0328269
	4	4.5935517	4.7574829	4.7470247	5.004821	4.4864205	5.6376553	4.0225737
	5	4.5768758	4.7178812	4.7694958	5.0001968	4.487845	5.6404632	4.0834364
	6	4.5768758	4.671277	4.6490001	5.0400789	4.5444411	5.6461698	4.0665999
	Average						**4.755602049**	

Table D.9: Throughput (Mbps) NS-3 experiment 1 hop

		2 HOPS					
		1 → 3	**2 → 4**	**3 → 5**	**4 → 6**	**5 → 7**	**6 → 8**
Nº of Trial	**1**	4.3376691	4.4314258	4.5964822	4.4593942	4.568592	4.2639712
	2	4.3724154	4.446179	4.6263005	4.4401233	4.5882921	4.4287793
	3	4.3790323	4.4539342	4.5595468	4.4510007	4.5869267	4.3714249
	4	4.3961813	4.4626792	4.6150586	4.4690173	4.5761477	4.4032104
	5	4.2850413	4.4033128	4.6195174	4.4449262	4.5489812	4.3632487
	6	4.4593902	4.5507871	4.6151698	4.3311868	4.5598771	4.2851125
	Average					**4.465287095**	

Table D.10: Throughput (Mbps) NS-3 experiment 2 hops

		3 HOPS				
		1 → 4	2 → 5	3 → 6	4 → 7	5 → 8
Nº of Trial	1	3.7679545	4.0046179	3.7571186	4.0478361	3.6876526
	2	3.8357442	3.9887356	3.9474263	4.0640167	3.650714
	3	3.7887933	4.0116615	3.9360032	3.9780138	3.6331466
	4	3.9144244	4.0452126	3.9232568	4.0205079	3.5991945
	5	3.885808	3.9594728	3.9776581	3.9856139	3.6075082
	6	3.9156657	3.9418078	3.9449253	4.0129368	3.701339
	Average				**3.884492229**	

Table D.11: Throughput (Mbps) NS-3 experiment 3 hops

		4 HOPS			
		1 → 5	2 → 6	3 → 7	4 → 8
Nº of Trial	1	3.3927363	3.4538957	3.4651162	3.1855374
	2	3.4034627	3.3724932	3.5420055	3.3026973
	3	3.434378	3.4356507	3.5197575	3.2148103
	4	3.4021741	3.4346603	3.6090711	3.3459754
	5	3.3806954	3.3721933	3.4930371	3.3206592
	6	3.4217972	3.4031172	3.5840392	3.3206592
	Average			**3.408775814**	

Table D.12: Throughput (Mbps) NS-3 experiment 4 hops

		5 HOPS		
		1 → 6	2 → 7	3 → 8
Nº of Trial	1	2.8413837	3.0770847	2.8741633
	2	2.8565449	3.1219793	2.7336691
	3	2.9367529	3.1379574	2.8856209
	4	2.8682988	3.1264579	2.8656552
	5	2.8861153	3.1071199	2.7493417
	6	2.8475779	3.0289456	2.7703916
	Average		**2.928614459**	

Table D.13: Throughput (Mbps) NS-3 experiment 5 hops

		6 HOPS	
		1 → 7	**2 → 8**
Nº of Trial	**1**	2.4779674	2.4362118
	2	2.6765218	2.3401383
	3	2.5936678	2.5418992
	4	2.6169669	2.3676781
	5	2.6252538	2.4193335
	6	2.6936232	2.3676781
	Average		**2.5130783**

Table D.14: Throughput (Mbps) NS-3 experiment 6 hops

		7 HOPS
		1 → 8
Nº of Trial	**1**	2.1882729
	2	2.1739403
	3	2.1895273
	4	2.1732374
	5	2.1363309
	6	2.1108662
	Average	**2.1620292**

Table D.15: Throughput (Mbps) NS-3 experiment 7 hops

APPENDIX E

List of Acronyms

AP	Access Point
BPSK	Binary Phase-Shift Keying
CCK	Complementary Code Keying
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
DIFS	Distributed coordination function Inter Frame Spacing
DSSS	Direct-Sequence Spread Spectrum
DTN	Data Transmission Network
GNU GPL	(GNU is Not Unix) General Public License
GTNetS	Global Traceability Networks
IEEE	Institute of Electrical and Electronics Engineers
IITP	Institute for Information Transmission Problems of the Russian Academy of Sciences
IP	Internet Protocol
ISM band	Industrial, Scientific and Medical radio bands
MAC	Media Access Control
NS-3	Network Simulator generation 3
OFDM	Orthogonal Frequency-Division Multiplexing
OS	Operative System
PCAP	Packet Capture
PDNS	Power Domain Name System
PhD	Doctor of Philosophy
PMP	Point-to-MultiPoint
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying
RSS	Received Signal Strength
RTT	Round Trip Time
SIFS	Short Inter Frame Spacing

TCP	Transmission-Control-Protocol
UDP	User Datagram Protocol
WEP	Wired Equivalent Privacy
Wi-Fi	Wireless Fidelity
WLAN	Wireless Local Area Networks
WMAN	Wireless Metropolitan Area Networks
WMN	Wireless Mesh Networks
WPA	Wi-Fi Protected Access
WPAN	Wireless Personal Area Networks

REFERENCES

[1] "IEEE Standard for Information technology, telecommunications and information exchange between systems local and metropolitan area networks specific requirements." http://standards.ieee.org/getieee802/download/802.11-2007.pdf. Last time acceded: 2010-02-15.

[2] Lee, M.J.; JianLiang Zheng; Young-Bae Ko; Shrestha, D.M., *Emerging starndars for wireless mesh technology. Wireless Communications*. Page(s): 56-63: IEEE.Volume 13, Issue 2, April 2006.

[3] "IEEE - the world's leading professional association for the advancement of technology." http://www.ieee.org/portal/site. Last time acceded: 2009-12-10.

[4] Gupta, P.;Kumar, P.R., *The capacity of wireless networks. Information Theory, IEEE Transactions on*. Page(s): 388-404: Addison-Wesley, March 2000.

[5] "Open80211s." http://o11s.org/. Last time acceded: 2009-10-14.

[6] "Ubuntu Home Page — Ubuntu." http://www.ubuntu.com/. Last time acceded: 2009-10-14.

[7] "About the IITP RAS." http://www.iitp.ru/en/about. Last time acceded: 2010-03-22.

[8] "The NS-3 Network Simulator." http://www.nsnam.org/index.html. Last time acceded: 2010-03-04.

[9] "Wireshark · Go deep." http://www.wireshark.org/. Last time acceded: 2009-10-14.

[10] "zd1211rw - Linux Wireless." http://linuxwireless.org/en/users/Drivers/zd1211rw. Last time acceded: 2009-10-14.

[11] "open80211s - Trac." http://o11s.org/trac#DriverStatus. Last time acceded: 2009-10-14.

[12] "ath9k - Linux Wireless." http://linuxwireless.org/en/users/Drivers/ath9k. Last time acceded: 2009-10-14.

[13] "Drivers - Linux Wireless." http://linuxwireless.org/en/users/Drivers. Last time acceded: 2009-10-14.

[14] "p54 - Linux Wireless." http://linuxwireless.org/en/users/Drivers/p54. Last time acceded: 2009-10-14.

[15] "D-Link High Speed 2.4GHz (801.11g) Wireless USB Adapter." http://www.dlink.com/products/?pid=334. Last time acceded: 2009-10-14.

[16] "IEEE P802.11 TGs." http://grouper.ieee.org/groups/802/11/Reports/tgs_update.htm. Last time acceded: 2009-12-29.

[17] "mac80211 - Linux Wireless." http://linuxwireless.org/en/developers/Documentation/mac80211. Last time acceded: 2009-12-29.

[18] "Download - Linux Wireless." http://wireless.kernel.org/en/users/Download#Archiveofcompat-wireless-2.6tarballs. Last time acceded: 2010-01-19.

[19] "git.kernel.org - linux/kernel/git/Linville/wireless-testing.git/summary." http://git.kernel.org/?p=linux/kernel/git/linville/wireless-testing.git;a=summary;. Last time acceded: 2010-01-19.

[20] "HOWTO - open80211s - Trac." http://o11s.org/trac/wiki/HOWTO. Last time acceded: 2010-02-10.

[21] Sachin Garg – Avaya Labs Research Basking Ridge, NJ USA; Martin Kappes – Avaya Labs Research Basking Ridge, NJ USA, "An Experimental Study of Throughput for UDP and VoIP Traffic in IEEE 802.11b Networks,"

[22] "802.11g (2.4GHz) Wireless USB 2.0 Adapter DWL-G122 D-Link AirPlus G TM Manual." https://www.cz.o2.com/public_conver/6a/55/cc/113539_142058_dwlg122_manual.pdf. Last time acceded: 2010-03-15.

[23] Theofilos Chrysikos, Giannis Georgopoulos, Stavros Kotsopoulos Wireless Telecommunications Laboratory – Department of Electrical & Computer Engineering – University of Patras, "Site-Specific Validation of ITU Indoor Path Loss Model at 2.4 GHz,"

[24] "NS-3: IEEE 802.11s draf." http://www.nsnam.org/doxygen-release/group__dot11s.html. Last time acceded: 2010-03-16.

[25] Douglas E. Comer, *Internetworking With TCP/IP VolI: Principles, Protocols and Architecture*. Page(s): 208-227: Third Edition.